1. Lecture Notes
    1. Syllabus
    2. Calendar
    3. Reading List
    4. Introduction to Operating Systems
    5. Process
    6. Synchronization, CPU Scheduling
    7. Memory management
    8. I/O devices and File systems
    9. Protection and Security
    10. Sample Exams
2. Projects
    1. Project 1: Exceptions and Simple System Calls
    2. Project 2: Multi-programming and Inter-process communication
    3. Project 3: Chat Application Using Nachos Networking Module
    4. Project 4: Emand Paging
    5. Project 5: Multi-programming, Inter-process Communication & Scheduling
    6. Project 6: Implementing File-System API
3. Review Questions
    1. Review question: Process
    2. Review question: Synchronization
    3. Review question: Memory management
    4. Review question: IO devices and File systems
    5. Review question: Protection and Security

Syllabus

Course Materials

Course Web page: http://www.vocw.edu.vn/content/m10103/latest/

Text: 'Operating Systems Concepts, 7th edition' by Silberschatz, Galvin and Gagne (ISBN 0471694665)

Description

This course will provide an introduction to operating system design and implementation. The operating system provides a well-known, convenient, and efficient interface between user programs and the bare hardware of the computer on which they run. The operating system is responsible for allowing resources (e.g., disks, networks, and processors) to be shared, providing common services needed by many different programs (e.g., file service, the ability to start or stop processes, and access to the printer), and protecting individual programs from one another.

The course will start with a brief historical perspective of the evolution of operating systems over the last fifty years, and then cover the major components of most operating systems. This discussion will cover the tradeoffs that can be made between performance and functionality during the design and implementation of an operating system. Particular emphasis will be given to three major OS subsystems: process management (processes, threads, CPU scheduling, synchronization, and deadlock), memory management (segmentation, paging, swapping), file systems, and operating system support for distributed systems.

Prerequisites

Programming with Data Structures

Architecture & Assembly Language, or equivalent, [In plain english: you need to have an understanding of computer architecture and have C++ programming skills]

Requirements and Grading

5 Homeworks (10%)

2 Programming Projects in C++ (30%)

2 Exams (60%)

- Exam 1: first 1/2 of course
- Exam 2: second 1/2 of course
- Each exam worth 30%

You are expected to attend class regularly, read the assigned reading before class, and participate in class discussions.

Late Policies

This course covers a lot of material and late assignments will seriously impact your ability to learn the next section of the course. Late programming assignments will be penalized 10\% per day, up to 4 days. Late homeworks will not be accepted (no exceptions). Please try to finish your assignments and homeworks on time.

Cooperation and Cheating

Feel free to discuss homework and labs with other members of the class, myself, or the TA. However, do not look at or copy another students solution to a homework or lab. I am not concerned with how you come to understand the problem and how to solve it, but once you have the

background necessary to solve it, you must provide your own solution. Exchanging homework or lab solutions is cheating and will be reported to the University, and you will lose credit for the course. Cheating will not be tolerated. A student found cheating on an exam will receive an automatic grade of F on the exam, and likely will fail the course as well.

Calendar

Week1	Introduction	Review C++
Week 2	OS and ArchitectureOperating System Structure	Project 1 hand outHomework 1 hand out
Week 3	Processes	Hw1 due, hw2 hand out
Week 4	Processes and Threads	
Week 5	Threads and CPU scheduling	Hw2 due
Week 6	CPU Scheduling and Synchronization	Hw 3 hand out
Week 7	Exam 1	Hw3 due
Week 8	Synchronization: Semaphores	Hw3 due Project 1 due, project2 hand out
Week 9	Synchronization: continue	Hw4 hand out
Week 10	Deadlocks and Deadlock Avoidance	

Week 11	Memory Management: Contiguous AllocationPaging and Segmented Paging	Hw4 due
Week 12	Page Replacement and approximations	Hw5 handout
Week 13	File System Interface File System Implementation	Hw5 due
Week 14	I/O Systems	Project 2 due
Week 15	Protection and Security	
Week 16	Exam 2	

Reading List

1. **1. Operating Systems: Design and Implementation (Second Edition), ISBN-10:** 0136386776, by Andrew S. Tanenbaum (Author), Albert S. Woodhull (Author)
2. **2. Applied Operating System Concepts, ISBN-10:** 0471365084, by Abraham Silberschatz (Author), Peter Baer Galvin (Author), Peter Galvin (Author), Avi Silberschatz (Author)
3. **3. Modern Operating Systems (2nd Edition), ISBN-10:** 0130313580, by Andrew S. Tanenbaum (Author)

Introduction to Operating Systems
Introduction to Operating Systems

Operating system is a hard term to define. Silberschatz e.t. defines that "**an operating system is a program that manages the computer hardware.**" However, what you consider an operating system depends on your needs and your view of the system. The first view is that operating system is a scheduler/simulator, on this view, the operating system has resources for which it is in charge, responsible for handing them out as well as recovering them later. Resources here include CPU, memory, I/O devices, and disk space. The second view is that operating system is a virtual machine, it means operating system provides a new machine. This machine could be the same as the underlying machine. Allows many users to believe they have an entire piece of hardware to themselves. This could implement a different, perhaps more powerful machine. Or just a different machine entirely. It may be useful to be able to completely simulate another machine with your current hardware. The other view is that operating system is a multiplexor which allows sharing of resources, provides protection from interference, and provides for a level of cooperation between users. This also for the economic reasons, we can't afford for all resources.

According to these three views, if we have enough hardware to give anyone too much; the hardware was well defined; the sharing problem is solved then we would not need operating systems. Unfortunatly, they will still be needed as a servant or provider of services: need to provide things like in the above views, but deal with environments that are less than perfect, need to help the users use the computer by: providing commonly used subroutines; providing access to hardware facilities; providing higher-level "abstract" facilities; providing an environment which is easy, pleasant, and productive to use. This view as a provider of services fits well with our modern network view of computing, where most resources are services.

Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. These goals form the basis for choices among various algorithms and strategies.

Because an operating system is large and complex, it must be created piece by piece. Each of these pieces should be well delineated portion of the system, with carefully defined inputs, outputs, and functions.

What are the desires of an operating system?

We can discuss them in term of: Usability, Facilities, Cost, and Adaptability.

Usability:

- Robustness

OS accept all valid inputs withour error, and gracefully handle all invalid inputs. OS would not be crashed in any circumtances, and could be recovered in the case that we suddenly remove hardware while they are running, or lost of power supplied.

- Consistency

For example, if "-" means options flags in one place, it means it in another. The key idea is **conventions**. We should base on the concept: The Principle of Least Astonishment. This helps us easy to understand and adapt with the new system.

- Proportionality Simple, cheap and frequent things are easy. Also, expensive and disastrous things are hard.
- Forgiving Errors can be recovered from. Reasonable error messages. Example from "rm"; UNIX vs. TOPS.
- Convenient Not necessary to repeat things, or do awkward procedures to accomplish things. Example copying a file took a batch job.
- Powerful Has high level facilities.

Facilities

- The system should supply sufficient for intended use
- The facilities is complete, don’t leave out any part of a facility.
- Appropriate, e.g. do not use fixed-width field input from terminal

Cost

- Want low cost and efficient services.
- Good algorithms. Make use of space/time tradeoffs, special hardware.
- Low overhead. Cost of doing nothing should be low. E.g., idle time at a terminal.
- Low maintenance cost. System should not require constant attention.

Adaptability

- Tailored to the environment. Support necessary activities. Do not impose unnecessary restrictions. What are the things people do most -- make them easy.
- Changeable over time. Adapt as needs and resources change. E.g., expanding memory and new devices, or new user population.
- Extendible-Extensible: Adding new facilities and features - which look like the old ones.

Two main perspectives of an operating system

- Outside view: whether your system can support for those kinds of program, do they have many facilities: compiler, database, do they easy to use! At this level we focus on what services the system provides.
- Inside view: related to the internals, code, data structures. This is the system programmer view of an operating system. At this level you understand not only what is provided, but how it is provided.

Five main components of an operating system

Process management

Process is a system abstraction, it illustrates that system has only one job to do. Every program running on a computer, be it background services or applications, is a process. As long as a von Neumann architecture is used to build computers, only one process per CPU can be run at a time. Older microcomputer OSes such as MS-DOS did not attempt to bypass this limit, with the exception of interrupt processing, and only one process could be run under them. Mainframe operating systems have had multitasking capabilities since the early 1960s. Modern operating systems enable concurrent execution of many processes at once via multitasking even with one CPU. Process management is an operating system's way of dealing with running multiple processes. Since most computers contain one processor with one core, multitasking is done by simply switching processes quickly. Depending on the operating system, as more processes run, either each time slice will become smaller or there will be a longer delay before each process is given a chance to run. Process management involves computing and distributing CPU time as well as other resources. Most operating systems allow a process to be assigned a priority which affects its allocation of CPU time. Interactive operating systems also employ some level of feedback in which the task with which the user is working receives higher priority. Interrupt driven processes will normally run at a very high priority. In many systems there is a background process, such as the System Idle Process in Windows, which will run when no other process is waiting for the CPU.

Memory management

Current computer architectures arrange the computer's memory in a hierarchical manner, starting from the fastest registers, CPU cache, random access memory and disk storage. An operating system's memory manager coordinates the use of these various types of memory by tracking which one is available, which is to be allocated or deallocated and how to move data between them. This activity, usually referred to as virtual memory management, increases the amount of memory available for each process by making the disk storage seem like main memory. There is a speed penalty

associated with using disks or other slower storage as memory – if running processes require significantly more RAM than is available, the system may start thrashing. This can happen either because one process requires a large amount of RAM or because two or more processes compete for a larger amount of memory than is available. This then leads to constant transfer of each process's data to slower storage.

Another important part of memory management is managing virtual addresses. If multiple processes are in memory at once, they must be prevented from interfering with each other's memory (unless there is an explicit request to utilise shared memory). This is achieved by having separate address spaces. Each process sees the whole virtual address space, typically from address 0 up to the maximum size of virtual memory, as uniquely assigned to it. The operating system maintains a page table that matches virtual addresses to physical addresses. These memory allocations are tracked so that when a process terminates, all memory used by that process can be made available for other processes.

The operating system can also write inactive memory pages to secondary storage. This process is called "paging" or "swapping" – the terminology varies between operating systems.

It is also typical for operating systems to employ otherwise unused physical memory as a page cache; requests for data from a slower device can be retained in memory to improve performance. The operating system can also pre-load the in-memory cache with data that may be requested by the user in the near future; SuperFetch is an example of this.

Disk and file systems

All operating systems include support for a variety of file systems.

Modern file systems comprise a hierarchy of directories. While the idea is conceptually similar across all general-purpose file systems, some differences in implementation exist. Two noticeable examples of this are the character used to separate directories, and case sensitivity.

Unix demarcates its path components with a slash (/), a convention followed by operating systems that emulated it or at least its concept of hierarchical directories, such as Linux, Amiga OS and Mac OS X. MS-DOS also emulated this feature, but had already also adopted the CP/M convention of using slashes for additional options to commands, so instead used the backslash (\) as its component separator. Microsoft Windows continues with this convention; Japanese editions of Windows use ¥, and Korean editions use ₩. Versions of Mac OS prior to OS X use a colon (:) for a path separator. RISC OS uses a period (.).

Unix and Unix-like operating allow for any character in file names other than the slash, and names are case sensitive. Microsoft Windows file names are not case sensitive.

File systems may provide journaling, which provides safe recovery in the event of a system crash. A journaled file system writes information twice: first to the journal, which is a log of file system operations, then to its proper place in the ordinary file system. In the event of a crash, the system can recover to a consistent state by replaying a portion of the journal. In contrast, non-journaled file systems typically need to be examined in their entirety by a utility such as fsck or chkdsk. Soft update is an alternative to journalling that avoids the redundant writes by carefully ordering the update operations. Log-structured file systems and ZFS also differ from traditional journaled file systems in that they avoid inconsistencies by always writing new copies of the data, eschewing in-place updates.

Many Linux distributions support some or all of ext2, ext3, ReiserFS, Reiser4, GFS, GFS2, OCFS, OCFS2, and NILFS. Linux also has full support for XFS and JFS, along with the FAT file systems, and NTFS.

Microsoft Windows includes support for FAT12, FAT16, FAT32, and NTFS. The NTFS file system is the most efficient and reliable of the four Windows file systems, and as of Windows Vista, is the only file system which the operating system can be installed on. Windows Embedded CE 6.0 introduced ExFAT, a file system suitable for flash drives.

Mac OS X supports HFS+ as its primary file system, and it supports several other file systems as well, including FAT16, FAT32, NTFS and ZFS.

Common to all these (and other) operating systems is support for file systems typically found on removable media. FAT12 is the file system most commonly found on floppy discs. ISO 9660 and Universal Disk Format are two common formats that target Compact Discs and DVDs, respectively. Mount Rainier is a newer extension to UDF supported by Linux 2.6 kernels and Windows-Vista that facilitates rewriting to DVDs in the same fashion as what has been possible with floppy disks.

Networking

Most current operating systems are capable of using the TCP/IP networking protocols. This means that one system can appear on a network of the other and share resources such as files, printers, and scanners using either wired or wireless connections.

Many operating systems also support one or more vendor-specific legacy networking protocols as well, for example, SNA on IBM systems, DECnet on systems from Digital Equipment Corporation, and Microsoft-specific protocols on Windows. Specific protocols for specific tasks may also be supported such as NFS for file access.

Security

Many operating systems include some level of security. Security is based on the two ideas that:

- The operating system provides access to a number of resources, directly or indirectly, such as files on a local disk, privileged system calls, personal information about users, and the services offered by the programs running on the system;
- The operating system is capable of distinguishing between some requesters of these resources who are authorized (allowed) to access the resource, and others who are not authorized (forbidden). While some systems may simply distinguish between "privileged" and "non-

privileged", systems commonly have a form of requester identity, such as a user name. Requesters, in turn, divide into two categories:

- Internal security: an already running program. On some systems, a program once it is running has no limitations, but commonly the program has an identity which it keeps and is used to check all of its requests for resources.
- External security: a new request from outside the computer, such as a login at a connected console or some kind of network connection. To establish identity there may be a process of authentication. Often a username must be quoted, and each username may have a password. Other methods of authentication, such as magnetic cards or biometric data, might be used instead. In some cases, especially connections from the network, resources may be accessed with no authentication at all.

In addition to the allow/disallow model of security, a system with a high level of security will also offer auditing options. These would allow tracking of requests for access to resources (such as, "who has been reading this file?").

Security of operating systems has long been a concern because of highly sensitive data held on computers, both of a commercial and military nature. The United StatesGovernment Department of Defense (DoD) created the Trusted Computer System Evaluation Criteria (TCSEC) which is a standard that sets basic requirements for assessing the effectiveness of security. This became of vital importance to operating system makers, because the TCSEC was used to evaluate, classify and select computer systems being considered for the processing, storage and retrieval of sensitive or classified information.

Operating systems Market share 2007

Client OS Market Share for June, 2007[1]
Windows XP - 81.94%
Windows Vista - 4.52%
Windows 2000 - 4.00%
Mac OS - 3.52%
MacIntel - 2.48%
Windows 98 - 1.14%
Linux - 0.71%
Windows NT - 0.66%
Windows Me - 0.59%
Nintendo Wii- 0.17%
Other - 0.20%

Questions to ask about operating systems.

Why are operating systems important?

- They consume more resources than any other program.

They may only use up a small percentage of the CPU time, but consider how many machines use the same program, all the time.

- They are the most complex programs.

They perform more functions for more users than any other program.

- They are necessary for any use of the computer.

When "the (operating) system" is down, the computer is down. Reliability and recovery from errors becomes critical.

- They are used by many users.

More hours of user time is spent dealing with the operating system. Visible changes in the operating system cause many changes to the users.

[1] [link] http://marketshare.hitslink.com/report.aspx?qprid=2&qpmr=15&qpdt=1&qpct=3&qptimeframe=M&qpsp=101

Why are operating systems difficult to create, use, and maintain?

- Size - too big for one person

Current systems have many millions lines of code. Involve 10-100 person years to build.

- Lifetime - the systems remain around longer than the programmers who wrote them.

The code is written and rewritten. Original intent is forgotten (UNIX was designed to be cute, little system - now 2 volumes this thick). Bug curve should be decreasing; but actually periodic - draw.

- Complexity - the system must do difficult things.

Deal with ugly I/O devices, multiplexing-juggling act, handle errors (**hard!**).

- Asynchronous - must do several things at once.

Handles interrupts, and must change what it is doing thousands of times a second - and still get work done.

- General purpose - must do many different things.

Run Doom, Java, Fortran, Lisp, Trek, Databases, Web Servers, etc. Everybody wants their stuff to run well.

History of Operating Systems

1. Single user (no OS).
2. Batch, uniprogrammed, run to completion.

    - The OS now must be protected from the user program so that it is capable of starting (and assisting) the next program in the batch.

3. Multiprogrammed

    - The purpose was to overlap CPU and I/O
    - Multiple batches

        - IBM OS/MFT (Multiprogramming with a Fixed number of Tasks)

            - OS for IBM system 360.
            - The (real) memory is partitioned and a batch is assigned to a fixed partition.
            - The memory assigned to a partition does not change.
            - Jobs were **spooled** from cards into the memory by a separate processor (an IBM 1401). Similarly output was spooled from the memory to a printer (a 1403) by the 1401.

        - IBM OS/MVT (Multiprogramming with a Variable number of Tasks) (then other names)

            - Each job gets just the amount of memory it needs. That is, the partitioning of memory changes as jobs enter and leave
            - MVT is a more “efficient” user of resources, but is more difficult.
            - When we study memory management, we will see that, with varying size partitions, questions like compaction and “holes” arise.

    - Time sharing

        - This is multiprogramming with rapid switching between jobs (processes). Deciding when to switch and which process to switch to is called scheduling.
        - We will study scheduling when we do processor management

4. Personal Computers

    - Serious PC Operating systems such as linux, Windows NT/2000/XP and (the newest) MacOS are multiprogrammed OSes.
    - GUIs have become important. Debate as to whether it should be part of the kernel.
    - Early PC operating systems were uniprogrammed and their direct descendants in some sense still are (e.g. Windows ME).

Operating systems are an unsolved problem.

Most of OS do not work very well, it crash too often, too slow, awkward to use, etc. Usually they do not do everything they were designed to do. They do not adapt to changes very well, e.g new devices, processors, applications. There are no perfect models to emulate.

Process
Process in Operating Systems

A **process** is an instance of a computer program that is being executed. While a program itself is just a passive collection of instructions, a process is the actual execution of those instructions. Several processes may be associated with the same program - each would execute independently (multithreading - where each thread represents a process), either synchronously (sequentially) or asynchronously (in parallel). Modern computer systems allow multiple programs and processes to be loaded into memory at the same time and, through time-sharing (or multitasking), give an appearance that they are being executed at the same time (concurrently) even if there is just one processor. Similarly, using a multithreading OS and/or computer architecture, parallel processes of the same program may actually execute simultaneously (on different CPUs) on a multiple CPU machine or network.[footnote]
http://en.wikipedia.org/wiki/Process_%28computing%29

Is a process the same as a program? No, it is both more and less. (what is a program? the statements that a user writes, or a command he invokes)

- More - a program is only part of the state; several processes may be derived from the same program. If I type "ls", something different happens than if you type it.
- Less - one program may use several processes, e.g. cc runs other things behind your back.

Some systems allow only one process. They are called uniprogramming systems (not uniprocessing; that means only one processor). Easier to write some parts of OS, but many other things are hard to do. E.g. compile a program in background while you edit another file; answer your phone and take messages while you are busy hacking.

Overview

In general, a computer system process consists of the following resources:

- An image of the executable machine code associated with a program.
- Memory (typically some region of virtual memory); which includes the executable code, process-specific data (input and output), a call stack (to keep track of active subroutines and/or other events), and a heap to hold intermediate computation data generated during run time.
- Operating system descriptors of resources that are allocated to the process, such as file descriptors (Unix terminology) or handles (Windows), and data sources and sinks.
- Security attributes, such as the process owner and the process' set of permissions (allowable operations).
- Processor state (context), such as the content of registers, physical memory addressing, etc. The state is typically stored in computer registers when the process is executing, and in memory otherwise.

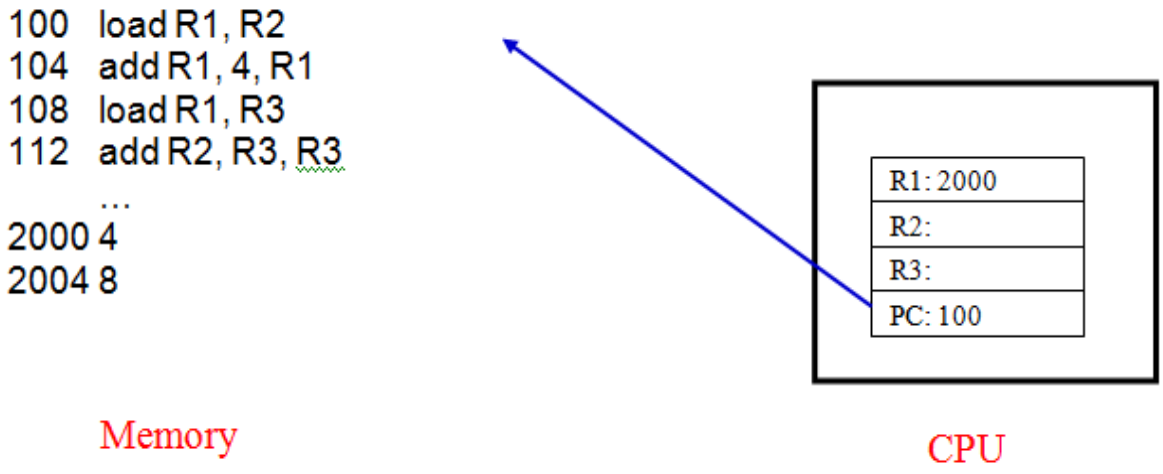
The operating system holds most of this information about active processes in data structures called process control blocks (PCB, part 3).

Any subset of resources, but typically at least the processor state, may be associated with each of the process' threads in operating systems that support threads or 'daughter' processes.

The operating system keeps its processes separated and allocates the resources they need so that they are less likely to interfere with each other and cause system failures (e.g., deadlock or thrashing). The operating system may also provide mechanisms for inter-process communication to enable processes to interact in safe and predictable ways.

Image of an executing program

An executing program



For example: we have a program as the following

```
public class foo {

    static private int yv = 0;
    static private int nv = 0;

    public static void main() {
        foo foo_obj = new foo;
        foo_obj->cheat();
    }

    public cheat() {
        int tyv = yv;
        yv = yv + 1;
        if (tyv < 10) {
            cheat();
        }
    }
}
```

The questions are:

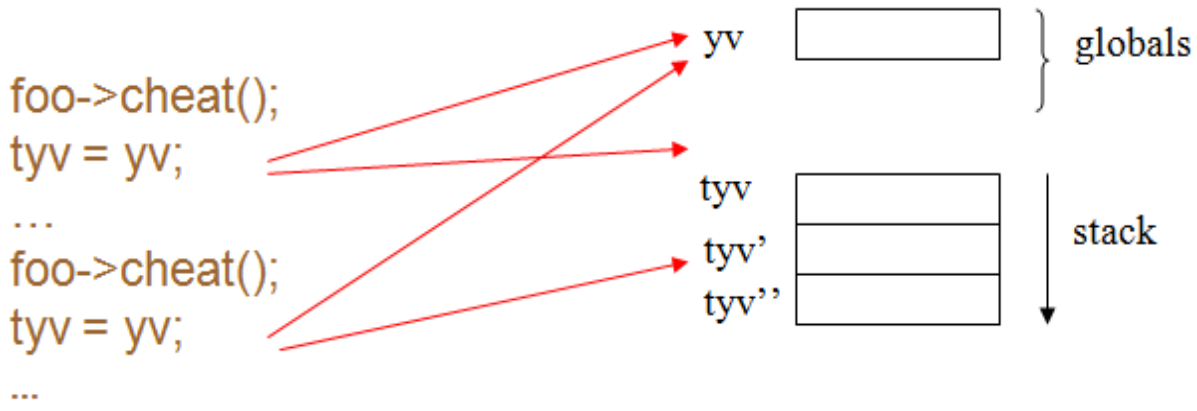
How to map a program like this to a Von Neuman machine?

Where to keep yv, nv?

What about foo_obj and tyv?

How to do foo->cheat()?

For the variable like yv, nv, we can easily give them some space on the memory. However, why can't we do the same for the local variable as tyv? Because while program is executing, we don't know it will invoke the procedure cheat() how many times, it means we don't know how much space we need to allocate for this variable. In this case, we will use stack, every time a new cheat() is called, the current tyv will be put in the stack, then we can pop it later when the program returns to this procedure. To find data allocated dynamically on stack, a stack pointer which always point at current activation record is used. We will discuss about activation record on section 2.2.



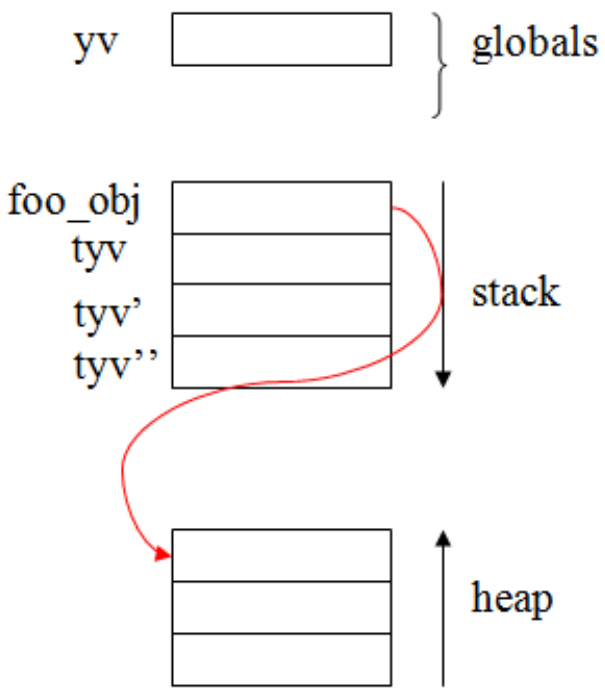
What about new objects? Of course, a memory location will be allocated for foo_obj. Is the stack an appropriate place to keep this object? Since this kind of object has many reusable code (all methods of class), we will waste a lot of our limited memory when we still use the stack for the new object. Heap is used in this case. Suppose we execute as the following:

```
yv = 0
nv = 0
main()
foo_obj = new foo

foo->cheat()
tyv = yv
yv = yv + 1

foo->cheat()
tyv = yv
yv = yv + 1

foo->cheat()
tyv = yv
yv = yv + 1
```



Activation record

Whenever you call a procedure there is certain information the program associates with that procedure call. The return address is a good example of some information the program maintains for a specific procedure call. Parameters and automatic local variables (i.e., those you declare in the VAR section) are additional examples of information the program maintains for each procedure call. Activation record is the term we'll use to describe the information the program associates with a specific call to a procedure.

Activation record is an appropriate name for this data structure. The program creates an activation record when calling (activating) a procedure and the data in the structure is organized in a manner identical to records. Perhaps the only thing unusual about an activation record (when comparing it to a standard record) is that the base address of the record is in the middle of the data structure, so you must access fields of the record at positive and negative offsets.

Construction of an activation record begins in the code that calls a procedure. The caller pushes the parameter data (if any) onto the stack. Then the execution of the CALL instruction pushes the return address onto the stack. At this point, construction of the activation record continues within in the procedure itself. The procedure pushes registers and other important state information and then makes room in the activation record for local variables. The procedure must also update the EBP register so that it points at the base address of the activation record.

To see what a typical activation record looks like, consider the following HLA procedure declaration:

```
procedure ARDemo( i:uns32; j:int32; k:dword );
nodisplay;
var
        a:int32;
        r:real32;
        c:char;
        b:boolean;
        w:word;
begin ARDemo;
        .
        .
        .
end ARDemo;
```

Whenever an HLA program calls this ARDemo procedure, it begins by pushing the data for the parameters onto the stack. The calling code will push the parameters onto the stack in the order they appear in the parameter list, from left to right. Therefore, the calling code first pushes the value for the i parameter, then it pushes the value for the j parameter, and it finally pushes the data for the k parameter. After pushing the parameters, the program calls the ARDemo procedure. Immediately upon entry into the ARDemo procedure, the stack contains these four items arranged as shown in Figure 3.1

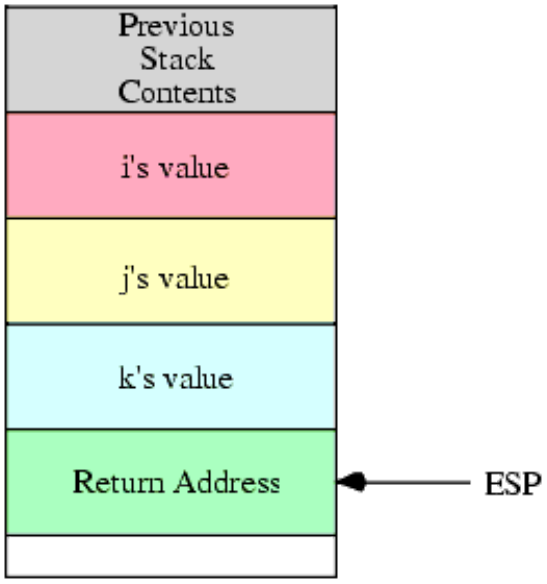


Figure 3.1 Stack Organization Immediately Upon Entry into ARDemo

The first few instructions in ARDemo (note that it does not have the @NOFRAME option) will push the current value of EBP onto the stack and then copy the value of ESP into EBP. Next, the code drops the stack pointer down in memory to make room for the local variables. This produces the stack organization shown in Figure 3.2

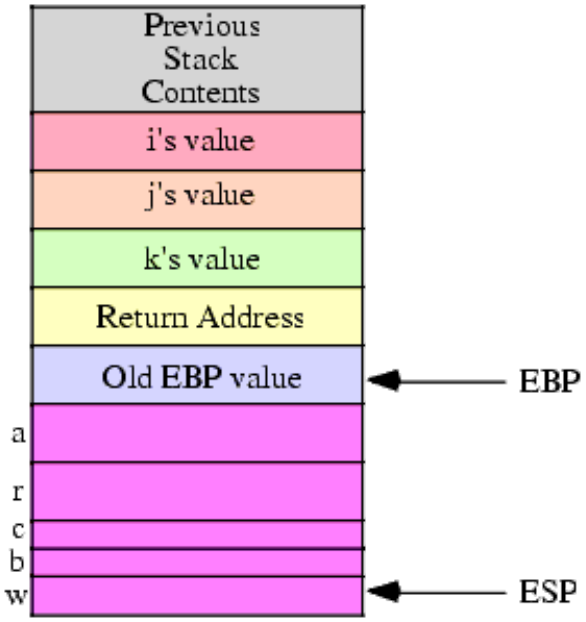


Figure 3.2 Activation Record for ARDemo

To access objects in the activation record you must use offsets from the EBP register to the desired object. The two items of immediate interest to you are the parameters and the local variables. You can access the parameters at positive offsets from the EBP register, you can access the local variables at negative offsets from the EBP register as Figure 3.3 shows:

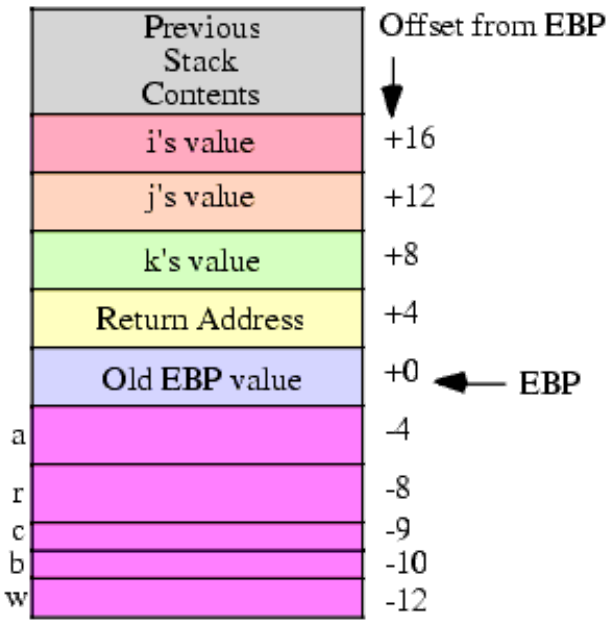


Figure 3.3 Offsets of Objects in the ARDemo Activation Record

Intel specifically reserves the EBP (extended base pointer) for use as a pointer to the base of the activation record. This is why you should never use the EBP register for general calculations. If you arbitrarily change the value in the EBP register you will lose access to the current procedure's parameters and local variables.

Threads

In modern operating systems, each process can have several threads. Multiple threads share the same program code, operating system resources (such as memory and file access) and operating system permissions (for file access as the process they belong to). A process that has only one thread is referred to as a single-threaded process, while a process with multiple threads is referred to as a multi-threaded process. Multi-threaded processes have the advantage that they can perform several tasks concurrently without the extra overhead needed to create a new process and handle synchronized communication between these processes. For example a word processor could perform a spell check as the user types, without freezing the application - one thread could handle user input, while another runs the spell checking utility

Process Control Block (PCB)

A Process Control Block (PCB) is a data structure in the operating system kernel containing the information needed to manage a particular process. The PCB is "the manifestation of a process in an operating system".

Included information

Implementations differ, but in general a PCB will include, directly or indirectly:

- The identifier of the process (a process identifier, or PID)
- Register values for the process including, notably,

the Program Counter value for the process

- The address space for the process
- Priority
- A list of open files & sockets
- Process accounting information, such as when the process was last run, how much CPU time it has accumulated, etc.
- Pointer to the next PCB i.e. pointer to the PCB of the next process to run

During a context switch, the running process is stopped and another process is given a chance to run. The kernel must stop the execution of the running process, copy out the values in hardware registers to its PCB, and update the hardware registers with the values from the PCB of the new process.

Location of the PCB

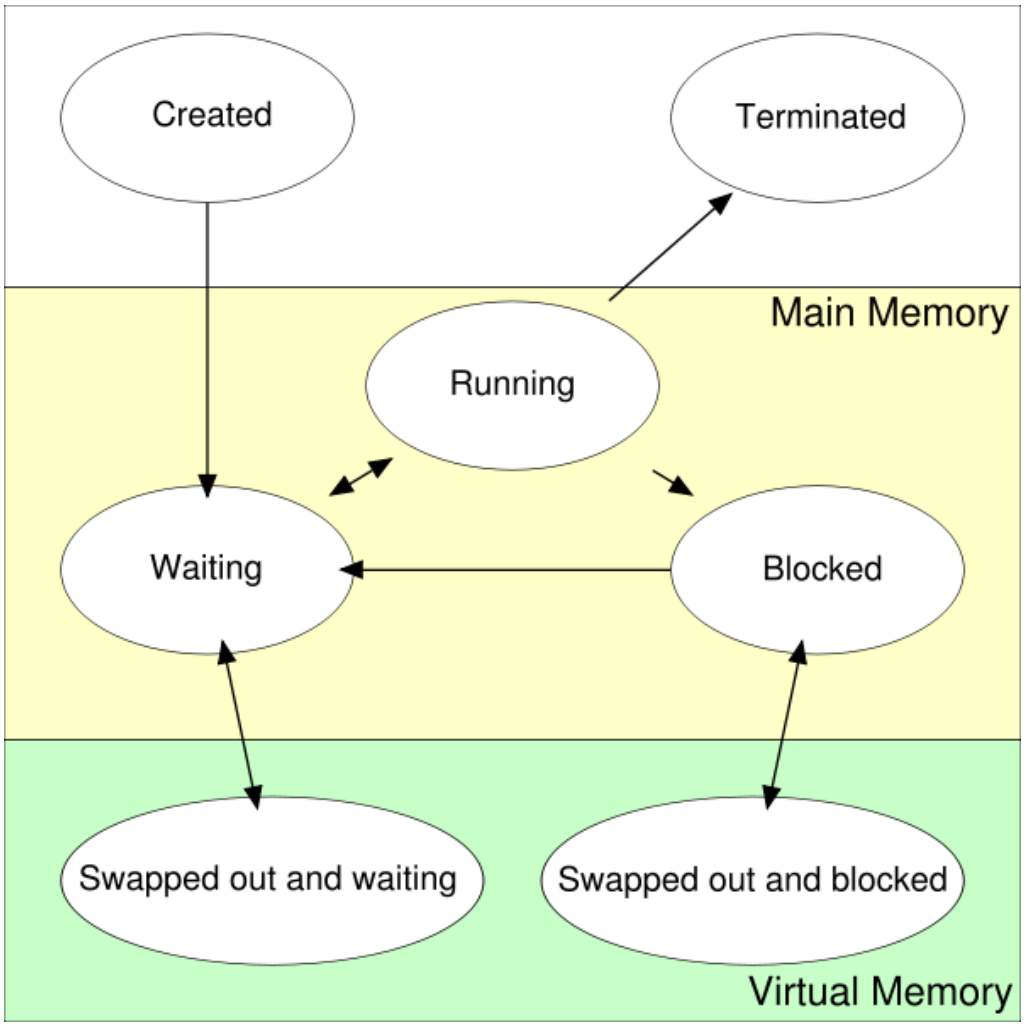
Since the PCB contains the critical information for the process, it must be kept in an area of memory protected from normal user access. In some operating systems the PCB is placed in the beginning of the kernel stack of the process since that is a convenient protected location.

Process states

Processes go through various process states which determine how the process is handled by the operating system kernel. The specific implementations of these states vary in different operating systems, and the names of these states are not standard, but the general high-level functionality is the same.

When a process is created, it needs to wait for the process scheduler to set its status to "waiting" and load it into main memory from secondary storage device (such as a hard disk or a CD-ROM). Once the process has been assigned to a processor by a short-term scheduler, a context switch is performed (loading the process into the processor) and the process state is set to "running" - where the processor executes its instructions. If a process needs to wait for a resource (such as waiting for user input, or waiting for a file to become available), it is moved into the "blocked" state until it no longer needs to wait - then it is moved back into the "waiting" state. Once the process finishes execution, or is terminated by the operating system, it is moved to the "terminated" state where it waits to be removed from main memory

Process states



Process states

Context switch

A context switch is the computing process of storing and restoring the state of a CPU such that multiple processes can share a single CPU resource. The context switch is an essential feature of a multitasking operating system. Context switches are usually computationally intensive and much of the design of operating systems is to optimize the use of context switches. A context switch can mean a register context switch, a task context switch, a

thread context switch, or a process context switch. What constitutes the context is determined by the processor and the operating system.

How is switching code invoked?

Preemptive: user thread executing ® clock interrupt ® PC modified by hardware to “vector” to interrupt handler ® user thread state is saved for restart ® clock interrupt handler is invoked ® disable interrupt checking ® check whether current thread has run “long enough” ® if yes, post asynchronous software trap (AST) ® enable interrupt checking ® exit interrupt handler ® enter “return-to-user” code ® check whether AST was posted ® if not, restore user thread state and return to executing user thread; if AST was posted, call context switch code

Non-preemptive: user thread executing ® system call to perform I/O ® user thread state is saved for restart ® OS code to perform system call is invoked ® I/O operation started (by invoking I/O driver) ® set thread status to waiting ® move thread’s TCB from run queue to wait queue associated with specific device ® call context switching code

Software vs hardware context switching

Intel 80386 and higher CPUs contain hardware support for context switches. However, most modern operating systems perform software context switching, which can be used on any CPU, rather than hardware context switching in an attempt to obtain improved performance. Software context switching was first implemented in Linux for Intel-compatible processors with the 2.4 kernel.

One major advantage claimed for software context switching is that, whereas the hardware mechanism saves almost all of the CPU state, software can be more selective and save only that portion that actually needs to be saved and reloaded. However, there is some question as to how important this really is in increasing the efficiency of context switching. Its

advocates also claim that software context switching allows for the possibility of improving the switching code, thereby further enhancing efficiency, and that it permits better control over the validity of the data that is being loaded.

The Cost of Context Switching

Context switching is generally computationally intensive. That is, it requires considerable processor time, which can be on the order of nanoseconds for each of the tens or hundreds of switches per second. Thus, context switching represents a substantial cost to the system in terms of CPU time and can, in fact, be the most costly operation on an operating system.

Consequently, a major focus in the design of operating systems has been to avoid unnecessary context switching to the extent possible. However, this has not been easy to accomplish in practice. In fact, although the cost of context switching has been declining when measured in terms of the absolute amount of CPU time consumed, this appears to be due mainly to increases in CPU clock speeds rather than to improvements in the efficiency of context switching itself.

One of the many advantages claimed for Linux as compared with other operating systems, including some other Unix-like systems, is its extremely low cost of context switching and mode switching.

Entering and Exiting the kernel

User and Kernel Address Spaces

In a modern operating system, each user process runs in its own address space, and the kernel operates in its protected space. At the processor level (machine code level), the main distinction between the kernel and a user process is the ability to access certain resources such as executing

privileged instructions, reading or writing special registers, and accessing certain memory locations.

The separation of user process from user process insures that each process will not disturb each other. The separation of user processes from the kernel insures that user processes will not be able to arbitrarily modify the kernel or jump into its code. It is important that processes cannot read the kernel's memory, and that it cannot directly call any function in the kernel. Allowing such operations to occur would invalidate any protection that the kernel wants to provide.

Operating systems provide a mechanism for selectively calling certain functions in the kernel. These select functions are called kernel calls or system calls, and act as gateways into the kernel. These gateways are carefully designed to provide safe functionality. They carefully check their parameters and understand how to move data from a user process into the kernel and back again. We will discuss this topic in more detail in the Memory Management section of the course.

The Path In and Out of the Kernel

The only way to enter the operating kernel is to generate a processor interrupt. Note the emphasis on the word "only". These interrupts come from several sources:

- I/O devices: When a device, such as a disk or network interface, completes its current operation, it notifies the operating system by generating a processor interrupt.
- Clocks and timers: Processors have timers that can be periodic (interrupting on a fixed interval) or count-down (set to complete at some specific time in the future). Periodic timers are often used to trigger scheduling decisions. For either of these types of timers, an interrupt is generated to get the operating system's attention.
- Exceptions: When an instruction performs an invalid operation, such as divide-by-zero, invalid memory address, or floating point overflow, the processor can generate an interrupt.

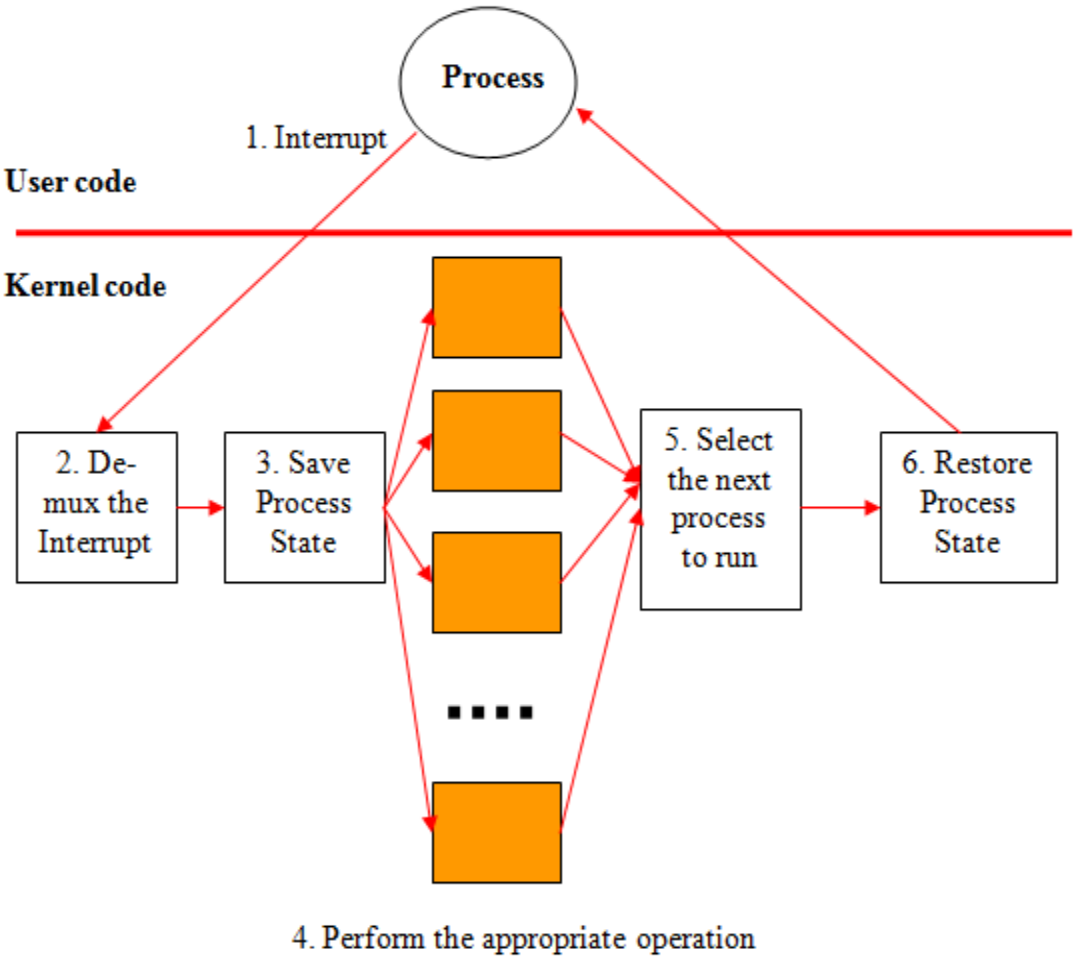
- Software Interrupts (Traps): Processors provide one or more instructions that will cause the processor to generate an interrupt. These instructions often have a small integer parameter. Trap instructions are most often used to implement system calls and to be inserted into a process by a debugger to stop the process at a breakpoint.

The flow of control is as follows (and illustrated below):

1. The general path goes from the executing user process to the interrupt handler. This step is like a forced function call, where the current PC and processor status are saved on a stack.
2. The interrupt handler decides what type of interrupt was generated and calls the appropriate kernel function to handle the interrupt.
3. The general run-time state of the process is saved (as on a context switch).
4. The kernel performs the appropriate operation for the system call. This step is the "real" functionality; all the steps before and after this one are mechanisms to get here from the user call and back again.
5. if the operation that was performed was trivial and fast, then the kernel returns immediately to the interrupted process. Otherwise, sometime later (it might be much later), after the operation is complete, the kernel executes its short-term scheduler (dispatcher) to pick the next process to run.

Note that one side effect of an interrupt might be to terminate the currently running process. In this case, of course, the current process will never be chosen to run next!

1. The state for the selected process is loaded into the registers and control is transferred to the process using some type of "return from interrupt" instruction.



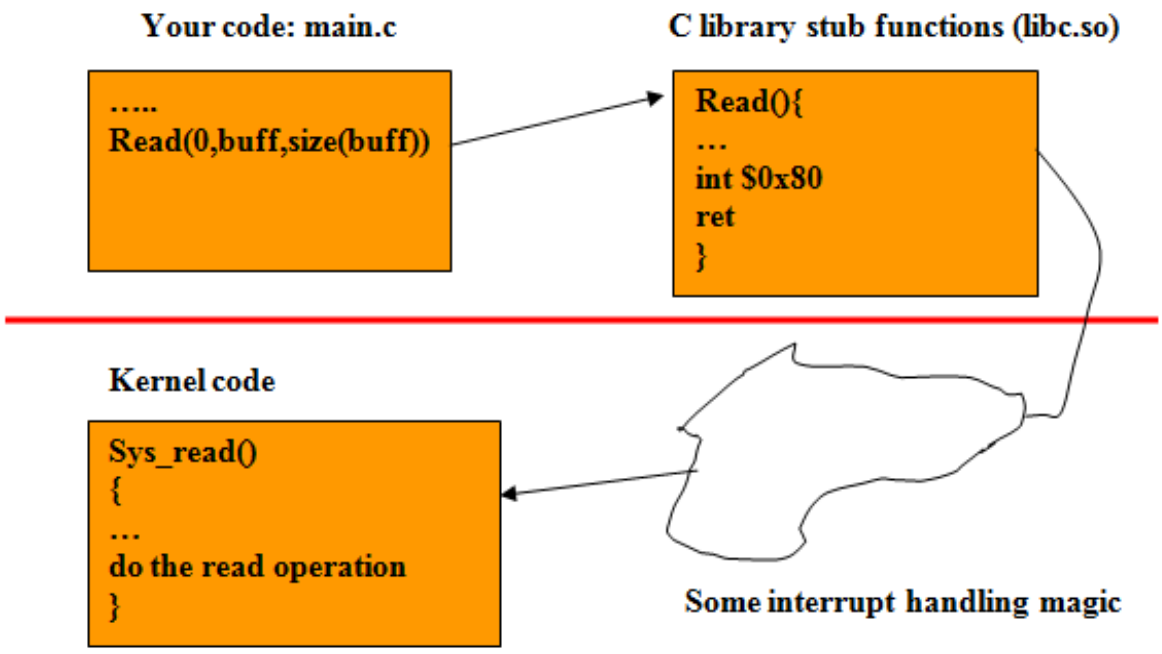
The system call path

One of the most important uses of interrupts, and one of the least obvious when you first study about operating systems, is the system call. In your program, you might request a UNIX system to read some data from a file with a call that looks like:

rv = read(0,buff,sizeof(buff));

Somewhere, deep down in the operating system kernel, is a function that implements this read operation. For example, in Linux, the routine is called sys_read().

The path from the simple read() function call in your program to the sys_read() routine in the kernel takes you through some interesting and crucial magic. The path goes from your code to a system call stub function that contains a trap or interrupts instruction, to an interrupt handler in the kernel, to the actual kernel function. The return path is similar, with the addition of some important interactions with the process dispatcher.



System Call Stub Functions

The system call stub functions provide a high-level language interface to a function whose main job is to generate the software interrupt (trap) needed to get the kernel's attention. These functions are often called wrappers.

The stub functions on most operating systems do the same basic steps. While the details of implementation differ, they include the following:

(1)

set up the parameters,

(2)

trap to the kernel,

(3)

check the return value when the kernel returns, and

(4)

(a)

if no error: return immediately, else

(b)

if there is an error: set a global error number variable (called "errno") and return a value of -1.

Below are annotated examples of this code from both the Linux (x86) and Solaris (SPARC) version of the C library. As an exercise, for the Linux and Solaris versions of the code, divide the code into the parts described above and label each part.

x86 Linux read (glibc 2.1.3)

```
read:       push   %ebx
            mov    0x10(%esp,1),%edx            ;
put the 3 parms in registers
            mov    0xc(%esp,1),%ecx
            mov    0x8(%esp,1),%ebx
            mov    $0x3,%eax                    ; 3
is the syscall # for read
            int    $0x80                        ;
trap to kernel
            pop    %ebx
            cmp    $0xfffff001,%eax             ;
```

```
check return value
            jae    read_err
read_ret:   ret                                ;
return if OK.
read_err:   push   %ebx
            call   read_next                   ;
push PC on stack
read_next:  pop    %ebx                        ;
pop PC off stack to %ebx
            xor    %edx,%edx                   ;
clear %edx
            add    $0x49a9,%ebx                ;
the following is a bunch of
            sub    %eax,%edx                   ;
...messy stuff that sets the
            push   %edx                        ;
...value fo the errno variable
            call   0x4000dfc0 <__errno_location>
            pop    %ecx
            pop    %ebx
            mov    %ecx,(%eax)
            or     $0xffffffff,%eax            ;
set return value to -1
            jmp    read_ret                    ;
return
```

SPARC Solaris 8

```
read:       st     %o0,[%sp+0x44]              !
save argument 1 (fd) on stack
read_retry: mov    3,%g1                       ! 3
is the syscall # for read
            ta     8                           !
trap to kernel
            bcc    read_ret                    !
branch if no error
            cmp    %o0,0x5b                    !
```

```
check for interrupt syscall
            be,a   read_retry                ! ...
and restart if so
            ld     [%sp+0x44],%o0            !
restore 1st param (fd)
            mov    %o7,%g1                   !
save return address
            call   read_next                 ! set
%o7 to PC
            sethi  %hi(0x1d800), %o5         ! the
following is a bunch of
read_next:  or     %o5, 0x10, %o5            !
...messy stuff that sets the
            add    %o5,%o7,%o5               !
...value of the errno variable
            mov    %g1, %o7                  !
...by calling _cerror.  also the
            ld     [%o5+0xe28],%o5           !
...return value is set to -1
            jmp    %o5
            nop
read_ret:   retl
            nop
```

Saving State and Invoking the Kernel Function

Below is a slightly simplified version of the Linux code that is called to handle a system call trap.

The first part of the code (starting at system_call) saves the registers of the user process and plays around with the memory management registers so that the kernel's internal data is accessible. It also finds the process table entry for this user process.

The trap instruction that caused the entry to the kernel has a parameter that specifies which system call is being invoked. The code starting at do_call

checks to see if this number is in range, and then calls the function associated with this system call number. When this function returns, the return value (stored in the eax register) is saved in the place where all the other user registers are stored. As a result, when control is transferred from the kernel back to the user process, the return value will be in the right place.

After the system call is complete, it is time to return to the user process. There are two choices at this point: (1) either return directly the the user process that made the system call or (2) go through the dispatcher to select the next process to run. ret_from_sys_call

```
system_call:
        #
        #----Save orig_eax: system call number
        #    used to distinguish process that
entered
        #    kernel via syscall from one that
entered
        #    via some other interrupt
        #
        pushl %eax

        #
        #----Save the user's registers
        #
        pushl %es
        pushl %ds
        pushl %eax
        pushl %ebp
        pushl %edi
        pushl %esi
        pushl %edx
        pushl %ecx
        pushl %ebx

        #
```

```
        #----Set up the memory segment registers
so that the kernel's
        #    data segment can be accessed.
        #
        movl $(__KERNEL_DS),%edx
        movl %edx,%ds
        movl %edx,%es

        #
        #----Load pointer to task structure in
EBX. The task structure
        #    resides below the 8KB per-process
kernel stack.
        #
        movl $-8192, %ebx
        andl %esp, %ebx

        #
        #----Check to see if system call number is
a valid one, then
        #    look-up the address of the kernel
function that handles this
        #    system call.
        #
do_call:
        cmpl $(NR_syscalls),%eax
        jae badsys
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)

        # Put return value in EAX of saved user
context
        movl %eax,EAX(%esp)

        #
        #----If we can return directly to the
user, then do so, else go to
        #    the dispatcher to select another
```

```
process to run.
        #
ret_from_sys_call:
        cli        # Block interrupts; iret
effectively re-enables them
        cmpl $0,need_resched(%ebx)
        jne reschedule

        # restore user context (including data
segments)
        popl %ebx
        popl %ecx
        popl %edx
        popl %esi
        popl %edi
        popl %ebp
        popl %eax
        popl %ds
        popl %es
        addl $4,%esp                    # ignore
orig_eax
        iret

reschedule:
        call SYMBOL_NAME(schedule)
        jmp ret_from_sys_call
```

Independent and Cooperating processes

Independent process

One that is independent of the rest of the universe.

- Its state is not shared in any way by any other process.
- Deterministic: input state alone determines results.
- Reproducible.

- Can stop and restart with no bad effects (only time varies). Example: program that sums the integers from 1 to i (input).

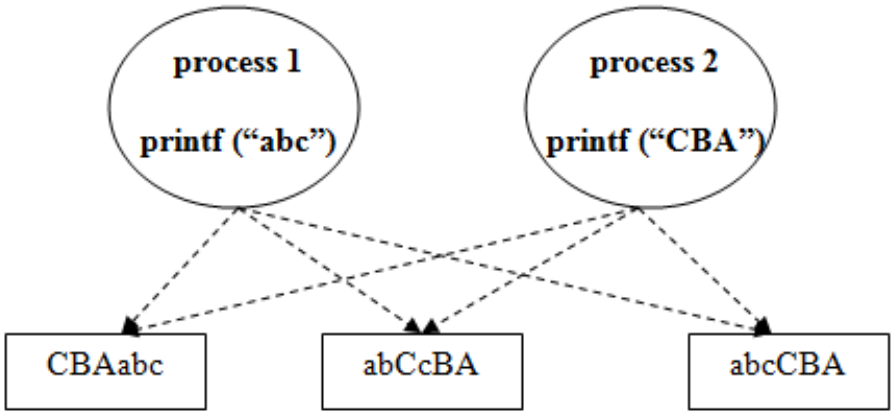
There are many different ways in which a collection of independent processes might be executed on a processor:

- Uniprogramming: a single process is run to completion before anything else can be run on the processor.
- Multiprogramming: share one processor among several processes. If no shared state, then order of dispatching is irrelevant.
- Multiprocessing: if multiprogramming works, then it should also be ok to run processes in parallel on separate processors.
    - A given process runs on only one processor at a time.
    - A process may run on different processors at different times (move state, assume processors are identical).
    - Cannot distinguish multiprocessing from multiprogramming on a very fine grain.

How often are processes completely independent of the rest of the universe?

Cooperating processes

- Machine must model the social structures of the people that use it. People cooperate, so machine must support that cooperation. Cooperation means shared state, e.g. a single file system.
- Cooperating processes are those that share state. (May or may not actually be "cooperating")
- Behavior is nondeterministic: depends on relative execution sequence and cannot be predicted a priori.
- Behavior is irreproducible.
- Example: one process writes "ABC", another writes "CBA". Can get different outputs, cannot tell what comes from which. E.g. which process output first "C" in "ABCCBA"? Note the subtle state sharing that occurs here via the terminal. Not just anything can happen, though. For example, "AABBCC" cannot occur.



When discussing concurrent processes, multiprogramming is as dangerous as multiprocessing unless you have tight control over the multiprogramming. Also bear in mind that smart I/O devices are as bad as cooperating processes (they share the memory).

Why permit processes to cooperate?

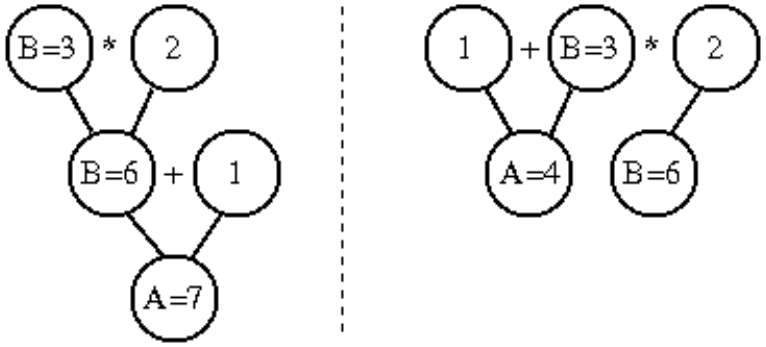
- Want to share resources:
    - One computer, many users.
    - One file of checking account records, many tellers. What would happen if there were a separate account for each teller? Could withdraw same money many times.
- Want to do things faster:
    - Read next block while processing current one.
    - Divide job into sub-jobs, execute in parallel.
- Want to construct systems in modular fashion. (e.g. tbl | eqn | troff)

Reading: Section 2.3.1 in Tanenbaum talks about similar stuff, but uses terms a little differently.

Basic assumption for cooperating process systems is that the order of some operations is irrelevant; certain operations are completely independent of

certain other operations. Only a few things matter:

- Example: A = 1; B = 2; has same result as B = 2; A = 1;
- Another example: A = B+1; B = 2*B cannot be re-ordered.



Race conditions: Suppose A=1 and A=2 are executed in parallel? Do not know what will happen; depends on which one goes fastest. What if they happen at EXACTLY the same time? Cannot tell anything without more information. Could end up with A=3!

Atomic operations: Before we can say ANYTHING about parallel processes, we must know that some operation is atomic, i.e. that it either happens in its entirety without interruption, or not at all. Cannot be interrupted in the middle. E.g. suppose that println is atomic -- what happens in println("ABC"); println("BCA") example?

- References and assignments are atomic in almost all systems. A=B will always get a good value for B, will always set a good value for A (not necessarily true for arrays, records, or even floating-point numbers).
- In uniprocessor systems, anything between interrupts is atomic.
- If you do not have an atomic operation, you cannot make one. Fortunately, the hardware folks give us atomic ops.

In fact, if there is true concurrency, it is very hard to make a perfect atomic operation; most of the time we settle for things that only work "most" of the

time.

- If you have any atomic operation, you can use it to generate higher-level constructs and make parallel programs work correctly. This is the approach we will take in this course.

Synchronization, CPU Scheduling
Synchronization, CPU Scheduling

Race Conditions

A **race condition** occurs when two (or more) processes are about to perform some action. Depending on the exact timing, one or other goes first. If one of the processes goes first, everything works, but if another one goes first, an error, possibly fatal, occurs.

Imagine two processes both accessing x, which is initially 10.

- One process is to execute x x+1
- The other is to execute x x-1
- When both are finished x should be 10
- But we might get 9 and might get 11!
- Show how this can happen (x x+1 is not atomic)
- Tanenbaum shows how this can lead to disaster for a printer spooler

Critical sections

We must prevent interleaving sections of code that need to be atomic with respect to each other. That is, the conflicting sections need **mutual exclusion**. If process A is executing its critical section, it excludes process B from executing its critical section. Conversely if process B is executing is critical section, it excludes process A from executing its critical section.

Requirements for a critical section implementation.

1. No two processes may be simultaneously inside their critical section.
2. No assumption may be made about the speeds or the number of CPUs.
3. No process outside its critical section (including the entry and exit code)may block other processes.
4. No process should have to wait forever to enter its critical section.

    - I do **NOT** make this last requirement.
    - I just require that the system as a whole make progress (so not all processes are blocked).
    - I refer to solutions that do not satisfy Tanenbaum's last condition as unfair, but nonetheless correct, solutions.
    - Stronger fairness conditions have also been defined.

Mutual exclusion with busy waiting

The operating system can choose not to preempt itself. That is, we do not preempt system processes (if the OS is client server) or processes running in system mode (if the OS is self

service). Forbidding preemption for system processes would prevent the problem above where x x+1 not being atomic crashed the printer spooler if the spooler is part of the OS.

But simply forbidding preemption while in system mode is not sufficient.

- Does not work for user-mode programs. So the Unix print spooler would not be helped.
- Does not prevent conflicts between the main line OS and interrupt handlers.
    - This conflict could be prevented by **disabling interrupts** while the main line is in its critical section.
    - Indeed, disabling (a.k.a. temporarily preventing) interrupts is often done for exactly this reason.
    - Do not want to block interrupts for too long or the system will seem unresponsive.

Does not work if the system has several processors.

- Both main lines can conflict.
- One processor cannot block interrupts on the other.

Software solutions for two processes

Initially P1wants=P2wants=false

Code for P1 Code for P2

Loop forever { Loop forever {

P1wants = true ENTRY P2wants = true

while (P2wants) {} ENTRY while (P1wants) {}

critical-section critical-section

P1wants = false EXIT P2wants = false

non-critical-section } non-critical-section }

Explain why this works.

But it is wrong! Why? (in case P1wants = P2wants = true then deadlock occurs)

Let's try again. The trouble was that setting want before the loop permitted us to get stuck. We had them in the wrong order!

Initially P1wants=P2wants=false

Code for P1 Code for P2

Loop forever { Loop forever {

while (P2wants) {} ENTRY while (P1wants) {}

P1wants = true ENTRY P2wants = true

critical-section critical-section

P1wants = false EXIT P2wants = false

non-critical-section } non-critical-section }

Explain why this works.

But it is wrong again! Why? (both processes may enter the CS)

So let's be polite and really take turns. None of this wanting stuff.

Initially turn=1

Code for P1 Code for P2

Loop forever { Loop forever {

while (turn = 2) {} while (turn = 1) {}

critical-section critical-section

turn = 2 turn = 1

non-critical-section } non-critical-section }

This one forces alternation, so is not general enough. Specifically, it does not satisfy condition three, which requires that no process in its non-critical section can stop another process from entering its critical section. With alternation, if one process is in its non-critical section (NCS) then the other can enter the CS once but not again.

The first example violated rule 4 (the whole system blocked). The second example violated rule 1 (both in the critical section. The third example violated rule 3 (one process in the NCS stopped another from entering its CS).

In fact, it took years (way back when) to find a correct solution. Many earlier “solutions” were found and several were published, but all were wrong. The first correct solution was found by a mathematician named Dekker, who combined the ideas of turn and wants. The basic idea is that you take turns when there is contention, but when there is no contention, the requesting process can enter. It is very clever, but I am skipping it (I cover it when I teach distributed operating systems in V22.0480 or G22.2251). Subsequently, algorithms with

better fairness properties were found (e.g., no task has to wait for another task to enter the CS twice).

What follows is Peterson's solution, which also combines turn and wants to force alternation only when there is contention. When Peterson's solution was published, it was a surprise to see such a simple soluntion. In fact Peterson gave a solution for any number of processes. A proof that the algorithm satisfies our properties (including a strong fairness condition) for any number of processes can be found in Operating Systems Review Jan 1990, pp. 18-22.

Initially P1wants=P2wants=false and turn=1

Code for P1 Code for P2

Loop forever { Loop forever {

P1wants = true P2wants = true

turn = 2 turn = 1

while (P2wants and turn=2) {} while (P1wants and turn=1) {}

critical-section critical-section

P1wants = false P2wants = false

non-critical-section non-critical-section

}}

Hardware assist (test and set)

TAS(b), where b is a binary variable, ATOMICALLY sets b = true and returns the OLD value of b.

Of course it would be silly to return the new value of b since we know the new value is true.

The word **atomically** means that the two actions performed by TAS(x) (testing, i.e., returning the old value of x and setting, i.e., assigning true to x) are inseparable. Specifically it is not possible for two concurrent TAS(x) operations to both return false (unless there is also another concurrent statement that sets x to false).

With TAS available implementing a critical section for any number of processes is trivial.

loop forever {

while (TAS(s)) {} ENTRY

CS

s = false EXIT

NCS

}

Sleep and Wakeup

Remark: Tanenbaum does both busy waiting (as above) and blocking (process switching) solutions. We will only do busy waiting, which is easier. Sleep and Wakeup are the simplest blocking primitives. Sleep voluntarily blocks the process and wakeup unblocks a sleeping process. We will not cover these.

Question: Explain the difference between busy waiting and blocking process synchronization.

Semaphores

Remark: Tannenbaum use the term semaphore only for blocking solutions. I will use the term for our busy waiting solutions. Others call our solutions spin locks.

P and V and Semaphores

The entry code is often called P and the exit code V. Thus the critical section problem is to write P and V so that

loop forever

P

critical-section

V

non-critical-section

satisfies

1. Mutual exclusion.
2. No speed assumptions.
3. No blocking by processes in NCS.
4. Forward progress (my weakened version of Tanenbaum's last condition).

Note that I use indenting carefully and hence do not need (and sometimes omit) the braces {} used in languages like C or java.

A **binary semaphore** abstracts the TAS solution we gave for the critical section problem.

- A binary semaphore S takes on two possible values “open” and “closed”.
- Two operations are supported
- P(S) is
- while (S=closed) {}
- S<--closed <== This is NOT the body of the while

where finding S=open and setting S<--closed is atomic

- That is, wait until the gate is open, then run through and atomically close the gate
- Said another way, it is not possible for two processes doing P(S) simultaneously to both see S=open (unless a V(S) is also simultaneous with both of them).
- V(S) is simply S<--open

The above code is not real, i.e., it is not an implementation of P. It is, instead, a definition of the effect P is to have.

To repeat: for any number of processes, the critical section problem can be solved by

loop forever

P(S)

CS

V(S)

NCS

The only specific solution we have seen for an arbitrary number of processes is the one just above with P(S) implemented via test and set.

Remark: Peterson's solution requires each process to know its processor number. The TAS soluton does not. Moreover the definition of P and V does not permit use of the processor number. Thus, strictly speaking Peterson did not provide an implementation of P and V. He did solve the critical section problem.

To solve other coordination problems we want to extend binary semaphores.

- With binary semaphores, two consecutive Vs do not permit two subsequent Ps to succeed (the gate cannot be doubly opened).
- We might want to limit the number of processes in the section to 3 or 4, not always just 1.

Both of the shortcomings can be overcome by not restricting ourselves to a binary variable, but instead define a **generalized** or **counting** semaphore.

- A counting semaphore S takes on non-negative integer values

- Two operations are supported
- P(S) is
- while (S=0) {}
- S--

where finding S>0 and decrementing S is atomic

- That is, wait until the gate is open (positive), then run through and atomically close the gate one unit
- Another way to describe this atomicity is to say that it is not possible for the decrement to occur when S=0 and it is also not possible for two processes executing P(S) simultaneously to both see the same necessarily (positive) value of S unless a V(S) is also simultaneous.
- V(S) is simply S++

These counting semaphores can solve what I call the semi-critical-section problem, where you premit up to k processes in the section. When k=1 we have the original critical-section problem.

initially S=k

loop forever

P(S)

SCS <== semi-critical-section

V(S)

NCS

Producer-consumer problem

- Two classes of processes
    - Producers, which produce times and insert them into a buffer.
    - Consumers, which remove items and consume them.

- What if the producer encounters a full buffer?Answer: It waits for the buffer to become non-full.
- What if the consumer encounters an empty buffer?Answer: It waits for the buffer to become non-empty.
- Also called the **bounded buffer** problem.
    - Another example of active entities being replaced by a data structure when viewed at a lower level (Finkel's level principle).

Initially e=k, f=0 (counting semaphore); b=open (binary semaphore)

Producer Consumer

loop forever loop forever

produce-item P(f)

P(e) P(b); take item from buf; V(b)

P(b); add item to buf; V(b) V(e)

V(f) consume-item

- k is the size of the buffer
- e represents the number of empty buffer slots
- f represents the number of full buffer slots
- We assume the buffer itself is only serially accessible. That is, only one operation at a time.
    - This explains the P(b) V(b) around buffer operations
    - I use; and put three statements on one line to suggest that a buffer insertion or removal is viewed as one atomic operation.
    - Of course this writing style is only a convention, the enforcement of atomicity is done by the P/V.
- The P(e), V(f) motif is used to force “bounded alternation”. If k=1 it gives strict alternation.

Semaphore implementation

Unfortunately, it is rare to find hardware that implements P & V directly (or messages, or monitors). They all involve some sort of scheduling and it is not clear that scheduling stuff belongs in hardware (layering). Thus semaphores must be built up in software using some lower-level synchronization primitive provided by hardware.

Need a simple way of doing mutual exclusion in order to implement P's and V's. We could use atomic reads and writes, as in "too much milk" problem, but these are very clumsy.

Uniprocessor solution: Disable interrupts.

```
class semaphore {
    private int count;

    public semaphore (int init)
    {
        count = init;
```

```
    }

    public void P ()
    {
        while (1) {
              Disable interrupts;
              if (count > 0) {
                   count--;
                   Enable interrupts;
              } else {
                   Enable interrupts;
              }
        }
    }

    public void V ()
    {
        Disable interrupts;
        count++;
        Enable interrupts;
    }
}
```

What is wrong with this code?

Multiprocessor solution:

Step 1: when P fails, put process to sleep; on V just wake up everybody, processes all try P again.

Step 2: label each process with semaphore it's waiting for, then just wakeup relevant processes.

Step 3: just wakeup a single process.

Step 4: add a queue of waiting processes to the semaphore. On failed P, add to queue. On V, remove from queue.

Why can we get away with only removing one process from queue at a time?

There are several tradeoffs implicit here: how many processes in the system, how much queuing on semaphores, storage requirements, etc. The most important thing is to avoid busy-waiting.

Is it "busy-waiting" if we use the solution in step 1 above?

What do we do in a multiprocessor to implement P's and V's? Cannot just turn off interrupts to get low-level mutual exclusion.

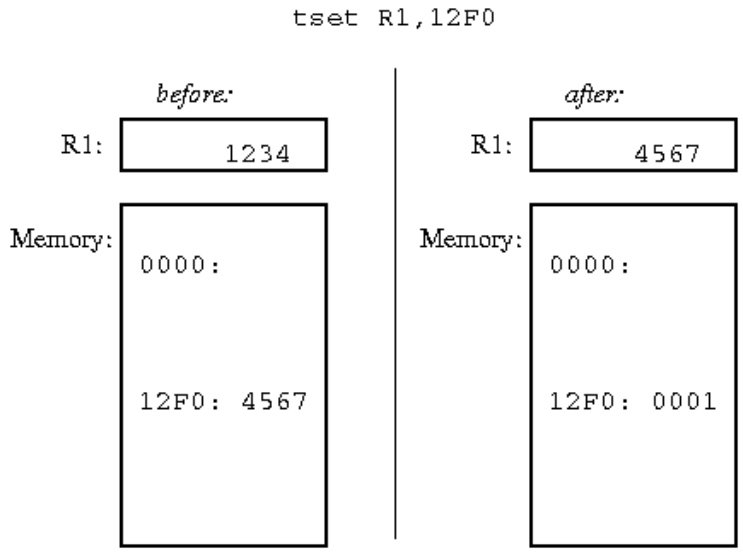
- Turn off all other processors?
- Use atomic “add item” and “take item”, as in "producer-consumer"?

In a multiprocessor, there must be busy-waiting at some level: cannot go to sleep if do not have mutual exclusion.

Most machines provide some sort of atomic read- modify-write instruction. Read existing value, store back in one atomic operation.

- E.g. Atomic increment.
- E.g. Test and set (IBM solution). Set value to one, but return OLD value. Use ordinary write to set back to zero.
- Read-modify-writes may be implemented directly in memory hardware, or in the processor by refusing to release the memory bus.

Using test and set for mutual exclusion: It is like a binary semaphore in reverse, except that it does not include waiting. 1 means someone else is already using it, 0 means it is OK to proceed. Definition of test and set prevents two processes from getting a 0->1 transition simultaneously.



Test and set is tricky to use. Using test and set to implement semaphores: For each semaphore, keep a test-and-set integer in addition to the semaphore integer and the queue of waiting processes.

```
class semaphore {
    private int t;
    private int count;
```

```
    private queue q;

    public semaphore(int init)
    {
        t = 0;
        count = init;
        q = new queue();
    }

    public void P()
    {
        Disable interrupts;
        while (TAS(t) != 0) { /* just spin */ };
        if (count > 0) {
            count--;
            t = 0;
            Enable interrupts;
            return;
        }
        Add process to q;
        t = 0;
        Enable interrupts;
        Redispatch;
    }

    public V()
    {
        Disable interrupts;
        while (TAS(t) != 0) { /* just spin */ };
        if (q == empty) {
            count++;
        } else {
            Remove first process from q;
            Wake it up;
        }
        t = 0;
        Enable interrupts;
    }
}
```

Why do we still have to disable interrupts in addition to using test and set?

Important point: implement some mechanism once, very carefully. Then always write programs that use that mechanism. Layering is very important.

Mutexes

Remark: Whereas we use the term semaphore to mean binary semaphore and explicitly say generalized or counting semaphore for the positive integer version, Tanenbaum uses semaphore for the positive integer solution and mutex for the binary version. Also, as indicated above, for Tanenbaum semaphore/mutex implies a blocking primitive; whereas I use binary/counting semaphore for both busy-waiting and blocking implementations. Finally, remember that in this course we are studying **only** busy-waiting solutions.

Monitors

Monitors are a high-level data abstraction tool combining three features:

- Shared data.
- Operations on the data.
- Synchronization, scheduling.

They are especially convenient for synchronization involving lots of state.

Existing implementations of monitors are embedded in programming languages. Best existing implementations are the Java programming language from Sun and the Mesa language from Xerox.

There is one binary semaphore associated with each monitor, mutual exclusion is implicit: P on entry to any routine, V on exit. This synchronization is automatically done by the compiler (because he makes automatic calls to the OS), and the programmer does not seem them. They come for free when the programmer declares a module to be a monitor.

Monitors are a higher-level concept than P and V. They are easier and safer to use, but less flexible, at least in raw form as above.

Probably the best implementation is in the Mesa language, which extends the simple model above with several additions to increase the flexibility and efficiency.

Do an example: implement a producer/consumer pair.

The "classic" Hoare-style monitor (using C++ style syntax):

```
class QueueHandler {

    private:
        static int BUFFSIZE = 200;
        int first;
        int last;
        int buff[BUFFSIZE];
        condition full;
```

```
    condition empty;

    int ModIncr(int v) {
        return (v+1)%BUFFSIZE;
    }

public:
    void QueueHandler (int);
    void AddToQueue (int);
    int RemoveFromQueue ();
};



void
QueueHandler::QueueHandler (int val)
{
    first = last = 0;
}

void
QueueHandler::AddToQueue (int val) {
{
    while (ModIncr(last) == first) {
        full.wait();
    }
    buff[last] = val;
    last = ModIncr(last);
    empty.notify();
}

int
QueueHandler::RemoveFromQueue ();
{
    while (first == last) {
        empty.wait();
    }
    int ret = buff[first];
    first = ModIncr(first);
    full.notify();
    return ret;
}
```

Java only allows one condition variable (implicit) per object. Here is the same solution in Java:

```
class QueueHandler {

        final static int BUFFSIZE = 200;
        private int first;
        private int last;
        private int buff[BUFFSIZE];


        private int ModIncr(int v) {
            return (v+1)%BUFFSIZE;
        }

        public QueueHandler (int val)
        {
            first = last = 0;
        }

        public synchronized void AddToQueue (int val) {
        {
            while (ModIncr(last) == first) {
                try { wait(); }
                catch (InterruptedException e) {}
            }
            buff[last] = val;
            last = ModIncr(last);
            notify();
        }

        public synchronized int RemoveFromQueue ();
        {
            while (first == last) {
                try { wait(); }
                catch (InterruptedException e) {}
            }
            int ret = buff[first];
            first = ModIncr(first);
            notify();
            return ret;
        }
```

Condition variables: things to wait on. Two types: (1) classic Hoare/Mesa condition variables and (2) Java condition variables.

Hoare/Mesa condition variables:

- condition.wait(): release monitor lock, put process to sleep. When process wakes up again, re-acquire monitor lock immediately.
- condition.notify(): wake up one process waiting on the condition variable (FIFO). If nobody waiting, do nothing.
- condition.broadcast(): wake up all processes waiting on the condition variable. If nobody waiting, do nothing.

Java condition variables:

- wait(): release monitor lock on current object; put thread to sleep.
- notify(): wake up one process waiting on the condition; this process will try to reacquire the monitor lock.
- notifyall(): wake up all processes waiting on the condition; each process will try to reacquire the monitor lock. (Of course, only one at a time will acquire the lock.)

Show how wait and notify solve the semaphore implementation problem. Mention that they can be used to implement any scheduling mechanism at all. How do wait and notify compare to P and V?

Do the readers' and writers' problem with monitors.

Summary:

- Not present in very many languages (yet), but extremely useful. Java is making monitors much more popular and well known.
- Semaphores use a single structure for both exclusion and scheduling, monitors use different structures for each.
- A mechanism similar to wait/notify is used internally to Unix for scheduling OS processes.
- Monitors are more than just a synchronization mechanism. Basing an operating system on them is an important decision about the structure of the entire system.

Message Passing

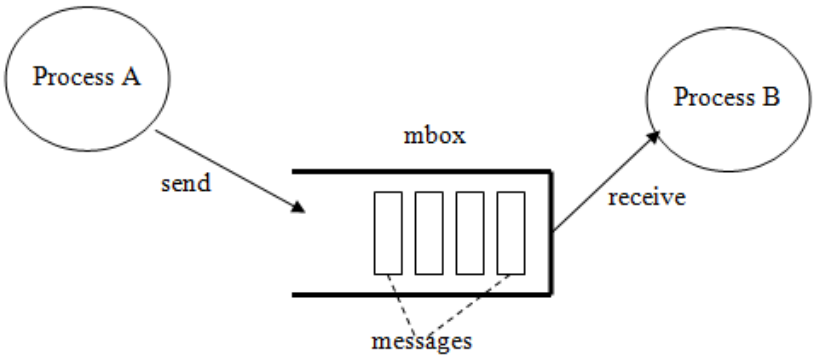
Up until now, discussion has been about communication using shared data. Messages provide for communication without shared data. One process or the other owns the data, never two at the same time.

Message = a piece of information that is passed from one process to another.

Mailbox = a place where messages are stored between the time they are sent and the time they are received.

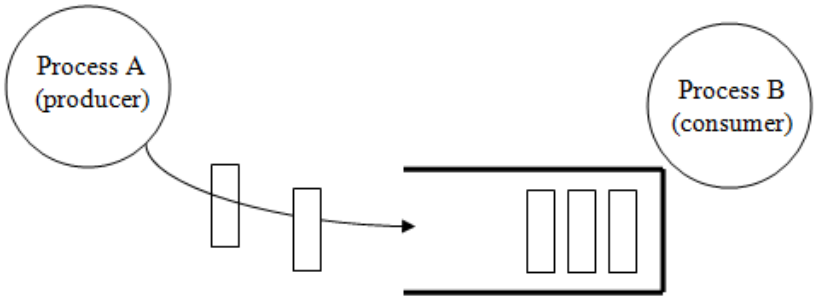
Operations:

- Send: place a message in a mailbox. If the mailbox is full, wait until there is enough space in the mailbox.
- Receive: remove a message from a mailbox. If the mailbox is empty, then wait until a message is placed in it.

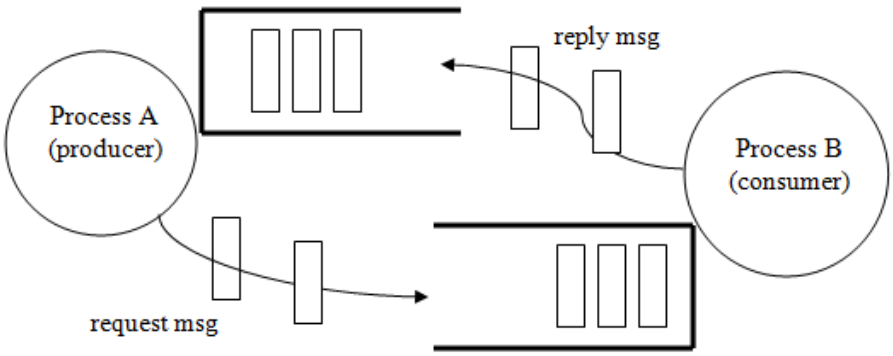


There are two general styles of message communication:

- 1-way: messages flow in a single direction (Unix pipes, or producer/consumer):



- 2-way: messages flow in circles (remote procedure call, or client/server):



Producer & consumer example:

Producer	**Consumer**
int buffer1[1000];while (1) {-- prepare buffer1 -- mbox.send(&buffer1);};	int buffer2[1000];while (1) {mbox.receive(&buffer2);-- process buffer2 --};

Note that buffer recycling is implicit, whereas it was explicit in the semaphore implementation.

Client & Server example:

Client	**Server**
int buffer1[1000];mbox1.send("read rutabaga");mbox2.receive(&buffer);	int buffer2[1000];int command[1000];mbox1.receive(&command);-- decode command ---- read file into buffer2 -- mbox2.send(&buffer2);

Note that this looks a lot like a procedure call&return. Explain the various analogs between procedure calls and message operations:

- Parameters:

- Result:
- Name of procedure:
- Return address:

Why use messages?

- Many kinds of applications fit into the model of processing a sequential flow of information, including all of the Unix filters.
- The component parties can be totally separate, except for the mailbox:
    - Less error-prone, because no invisible side effects: no process has access to another's memory.
    - They might not trust each other (OS vs. user).
    - They might have been written at different times by different programmers who knew nothing about each other.
    - They might be running on different processors on a network, so procedure calls are out of the question.

Which are more powerful, messages or monitors?

Message systems vary along several dimensions:

- Relationship between mailboxes and processes:
    - One mailbox per process, use process name in send and receive (simple but restrictive) [RC4000].
    - No strict mailbox-process association, use mailbox name (can have multiple mailboxes per process, can pass mailboxes from process to process, but trickier to implement) [Unix].
- Extent of buffering:
    - Buffering (more efficient for large transfers when sender and receiver run at varying speeds).
    - None -- rendezvous protocols (simple, OK for call-return type communication, know that message was received).
- Conditional vs. unconditional ops:
    - Unconditional receive: return message if mailbox is not empty, otherwise wait until message arrives.
    - Conditional receive: return message if mailbox is not empty, otherwise return special "empty" value.
    - Unconditional send: wait until mailbox has space.
    - Conditional send: return "full" if no space in mailbox (message is discarded).

What happens with rendezvous protocols and conditional operations?

- Additional forms of waiting:
    - Almost all systems allow many processes to wait on the same mailbox at the same time. Messages get passed to processes in order.
    - A few systems allow each process to wait on several mailboxes at once. The process gets the first message to arrive on any of the mailboxes. This is actually quite useful (give Caesar as an example).
- Constraints on what gets passed in messages:
    - None: just a stream of bytes (Unix pipes).
    - Enforce message boundaries (send and receive in same chunks).
    - Protected objects (e.g. a token for a mailbox).

How would the following systems fall into the above classifications?

- Condition variables
- Unix pipes

Classical IPC Problems

The Dining Philosophers Problem

A classical problem from Dijkstra

- 5 philosophers sitting at a round table
- Each has a plate of spaghetti
- There is a fork between each two
- Need two forks to eat

What algorithm do you use for access to the shared resource (the forks)?

- The obvious solution (pick up right; pick up left) deadlocks.
- Big lock around everything serializes.
- Good code in the book.

The purpose of mentioning the Dining Philosophers problem without giving the solution is to give a feel of what coordination problems are like. The book gives others as well. We are skipping these (again this material would be covered in a sequel course).

The Readers and Writers Problem

- Two classes of processes.
    - Readers, which can work concurrently.

    - Writers, which need exclusive access.

- Must prevent 2 writers from being concurrent.
- Must prevent a reader and a writer from being concurrent.
- Must permit readers to be concurrent when no writer is active.
- Perhaps want fairness (e.g., freedom from starvation).
- Variants

- Writer-priority readers/writers.
- Reader-priority readers/writers.

Quite useful in multiprocessor operating systems and database systems. The “easy way out” is to treat all processes as writers in which case the problem reduces to mutual exclusion (P and V). The disadvantage of the easy way out is that you give up reader concurrency. Again for more information see the web page referenced above.

The barbershop problem

The original barbershop problem was proposed by Dijkstra. A variation of it appears in Silberschatz and Galvin’s Operating Systems Concepts. A barbershop consists of a waiting room with n chairs, and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

Scheduling

Until now we have talked about processes, from now on we will talk about resources, the things operated upon by processes. Resources range from cpu time to disk space to channel I/O time.

Resources fall into two classes:

- Preemptible: processor or I/O channel. Can take resource away, use it for something else, then give it back later.
- Non-preemptible: once given, it cannot be reused until process gives it back. Examples are file space, terminal, and maybe memory.

OS makes two related kinds of decisions about resources:

- Allocation: who gets what. Given a set of requests for resources, which processes should be given which resources in order to make most efficient use of the resources? Implication is that resources are not easily preemptible.

- Scheduling: how long can they keep it. When more resources are requested than can be granted immediately, in which order should they be serviced? Examples are processor scheduling (one processor, many processes), memory scheduling in virtual memory systems. Implication is that resource is preemptible.

CPU Scheduling

Processes may be in any one of three general scheduling states:

- Running.
- Ready. That is, waiting for CPU time. Scheduler and dispatcher determine transitions between this and running state.
- Blocked. Waiting for some other event: disk I/O, message, semaphore, etc. Transitions into and out of this state are caused by various processes.

There are two parts to CPU scheduling:

- The dispatcher provides the basic mechanism for running processes.
- The scheduler is a piece of OS code that decides the priorities of processes and how long each will run.

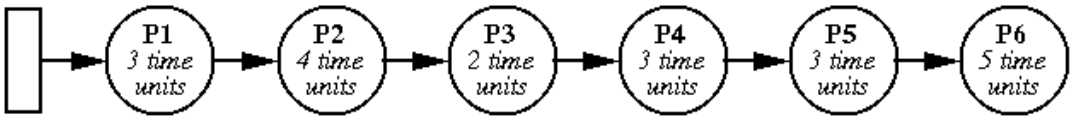
This is an example of policy/mechanism separation.

Goals for Scheduling Disciplines

- Efficiency of resource utilization (keep CPU and disks busy).
- Minimize overhead (context swaps).
- Minimize response time. (Define response time.)
- Distribute cycles equitably. What does this mean?

FCFS (also called FIFO)

Run until finished.

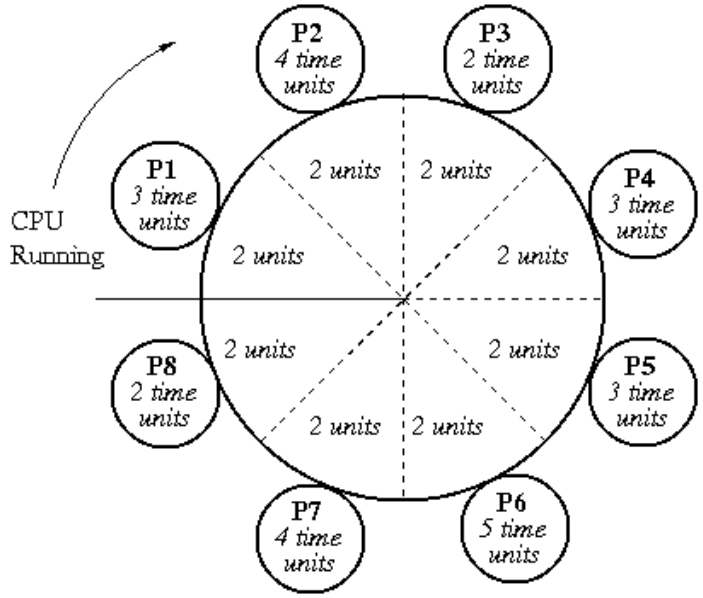


- In the simplest case this means uniprogramming.
- Usually, "finished" means "blocked". One process can use CPU while another waits on a semaphore. Go to back of run queue when ready.
- Problem: one process can monopolize CPU.

Solution: limit maximum amount of time that a process can run without a context switch. This time is called a time slice.

Round Robin

Run process for one time slice, then move to back of queue. Each process gets equal share of the CPU. Most systems use some variant of this. What happens if the time slice is not chosen carefully?



Originally, Unix had 1 sec. time slices. Too long. Most timesharing systems today use time slices of 10,000 - 100,000 instructions.

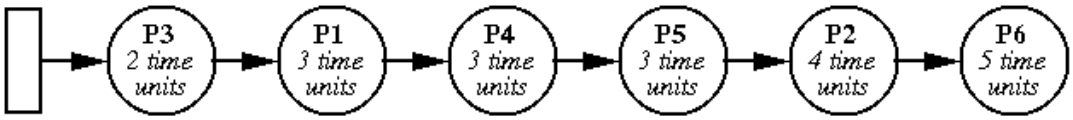
Implementation of priorities: run highest priority processes first, use round-robin among processes of equal priority. Re-insert process in run queue behind all processes of greater or equal priority.

Even round-robin can produce bad results occasionally. Go through example of ten processes each requiring 100 time slices.

What is the best we can do?

STCF

Shortest time to completion first with preemption. This minimizes the average response time.

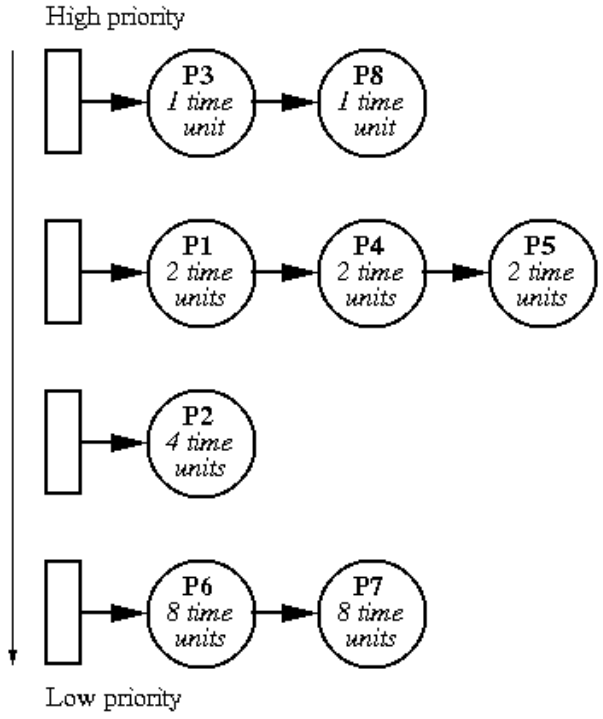


As an example, show two processes, one doing 1 ms computation followed by 10 ms I/O, one doing all computation. Suppose we use 100 ms time slice: I/O process only runs at 1/10th speed, effective I/O time is 100 ms. Suppose we use 1 ms time slice: then compute-bound process gets interrupted 9 times unnecessarily for each valid interrupt. STCF works quite nicely.

Unfortunately, STCF requires knowledge of the future. Instead, we can use past performance to predict future performance.

Exponential Queue (also called "multi-level feedback queues")

Attacks both efficiency and response time problems.



- Give newly runnable process a high priority and a very short time slice. If process uses up the time slice without blocking then decrease priority by 1 and double time slice for next time.
- Go through the above example, where the initial values are 1ms and priority 100.
- Techniques like this one are called adaptive. They are common in interactive systems.

- The CTSS system (MIT, early 1960's) was the first to use exponential queues.

Fair-share scheduling as implemented in Unix:

- Figure out each process' "share" of CPU, based on number of processes and priorities.
- Keep a history of recent CPU usage for each process: if it is getting less than its share, boost priority. If it is getting more than its share, reduce priority.
- Careful: could be unstable!

Summary:

- In principle, scheduling algorithms can be arbitrary, since the system should behave the same in any event.
- However, the algorithms have crucial effects on the behavior of the system:
    - Overhead: number of context swaps.
    - Efficiency: utilization of CPU and devices.
    - Response time: how long it takes to do something.
- The best schemes are adaptive. To do absolutely best, we would have to be able to predict the future.

Priority Inversion Problem

There are some curious interactions between scheduling and synchronization. A classic problem caused by this interaction was first observed in 1979 but Butler Lampson and David Redell at Xerox.

Suppose that you have three processes:

P1:	Highest priority
P2:	Medium priority
P3:	Lowest priority

And suppose that you have the following critical section, S:

S: mutex.P()

. . .

. . .

mutex.V()

The three processes execute as follows:

1. P3 enters S, locking the critical section.
2. P3 is preempted by the scheduler and P2 starts running.
3. P2 is preempted by the scheduler and P1 starts running.
4. P1 tries to enter S and is blocked at the P operation.
5. P2 starts running again, preventing P1 from running.

So, what's going wrong here? To really understand this situation, you should try to work out the example for yourself, before continuing to read.

- As long as process P2 is running, process P3 cannot run.
- If P3 cannot run, then it cannot leave the critical section S.
- If P3 does not leave the critical section, then P1 cannot enter.

As a result, P2 running (at medium priority) is blocking P1 (at highest priority) from running. This example is not an academic one. Many designers of real-time systems, where priority can be crucial, have stumbled over issue. You can read the original paper by Lampson and Redell to see their suggestion for handling the situation.

Memory management
Memory Management

Storage Allocation

Information stored in memory is used in many different ways. Some possible classifications are:

- Role in Programming Language:
    - Instructions (specify the operations to be performed and the operands to use in the operations).
    - Variables (the information that changes as the program runs: locals, owns, globals, parameters, dynamic storage).
    - Constants (information that is used as operands, but that never changes: pi for example).
- Changeability:
    - Read- only: (code, constants).
    - Read & write: (variables).

Why is identifying non-changing memory useful or important?

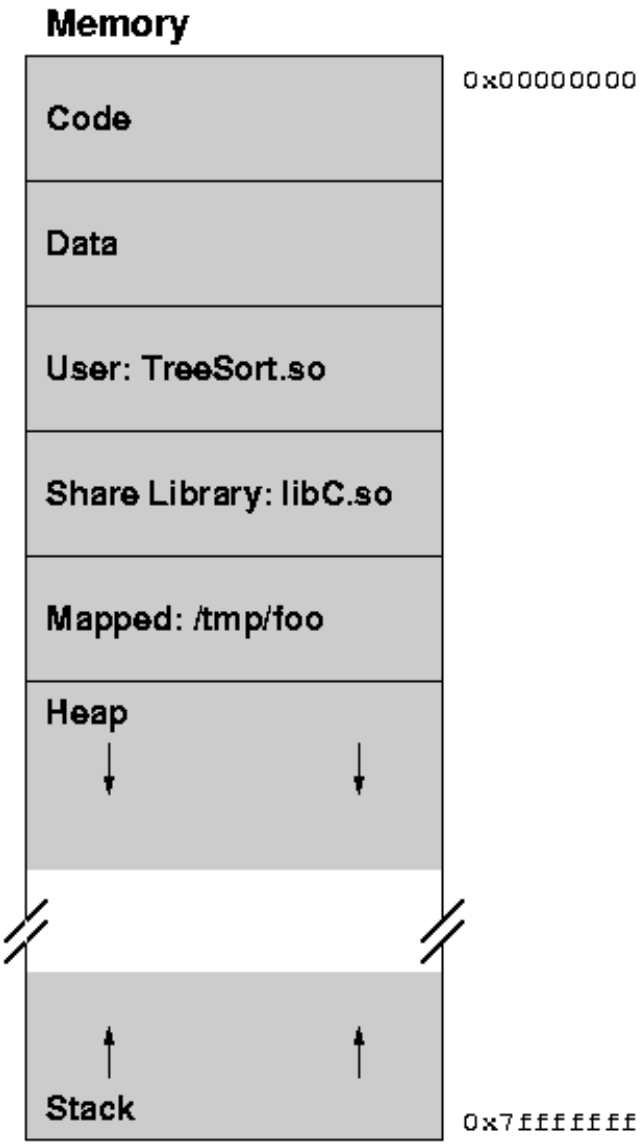
- Initialized:
    - Code, constants, some variables: yes.
    - Most variables: no.
- Addresses vs. Data: Why is this distinction useful or important?
- Binding time:
    - Static: arrangement determined once and for all, before the program starts running. May happen at compile-time, link-time, or load-time.
    - Dynamic: arrangement cannot be determined until runtime, and may change.

Note that the classifications overlap: variables may be static or dynamic, code may be read-only or read&write, etc.

The compiler, linker, operating system, and run-time library all must cooperate to manage this information and perform allocation.

When a process is running, what does its memory look like? It is divided up into areas of stuff that the OS treats similarly, called segments. In Unix, each process has three segments:

- Code (called "text" in Unix terminology)
- Initialized data
- Uninitialized data
- User's dynamically linked libraries (shared objects (.so) or dynamically linked libraries (.dll))
- Shared libraries (system dynamically linked libraries)
- Mapped files
- Stack(s)



In some systems, can have many different kinds of segments.

One of the steps in creating a process is to load its information into main memory, creating the necessary segments. Information comes from a file that gives the size and contents of each segment (e.g. a.out in Unix). The file is called an object file. See man 5 a.out for format of Unix object files.

Division of responsibility between various portions of system:

- Compiler: generates one object file for each source code file containing information for that file. Information is incomplete, since

each source file generally uses some things defined in other source files.
- Linker: combines all of the object files for one program into a single object file, which is complete and self-sufficient.
- Operating system: loads object files into memory, allows several different processes to share memory at once, provides facilities for processes to get more memory after they have started running.
- Run-time library: provides dynamic allocation routines, such as calloc and free in C.

Dynamic Memory Allocation

Why is not static allocation sufficient for everything? Unpredictability: cannot predict ahead of time how much memory, or in what form, will be needed:

- Recursive procedures. Even regular procedures are hard to predict (data dependencies).
- OS does not know how many jobs there will be or which programs will be run.
- Complex data structures, e.g. linker symbol table. If all storage must be reserved in advance (statically), then it will be used inefficiently (enough will be reserved to handle the worst possible case).

Need dynamic memory allocation both for main memory and for file space on disk.

Two basic operations in dynamic storage management:

- Allocate
- Free

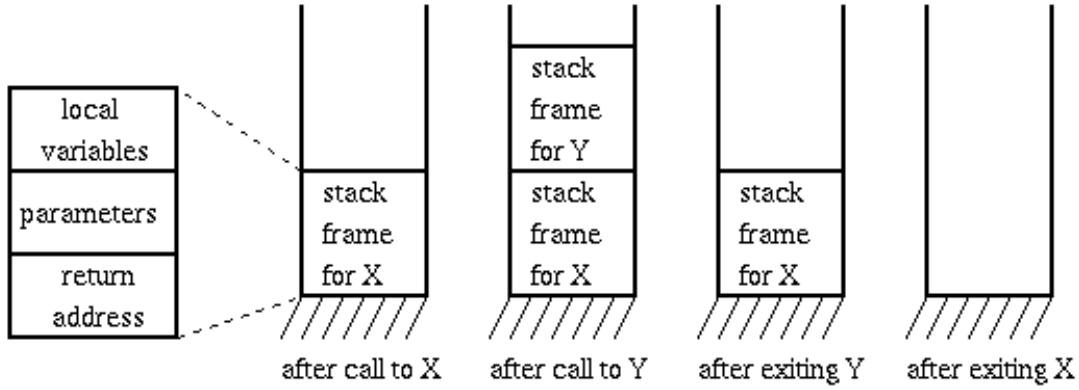
Dynamic allocation can be handled in one of two general ways:

- Stack allocation (hierarchical): restricted, but simple and efficient.
- Heap allocation: more general, but less efficient, more difficult to implement.

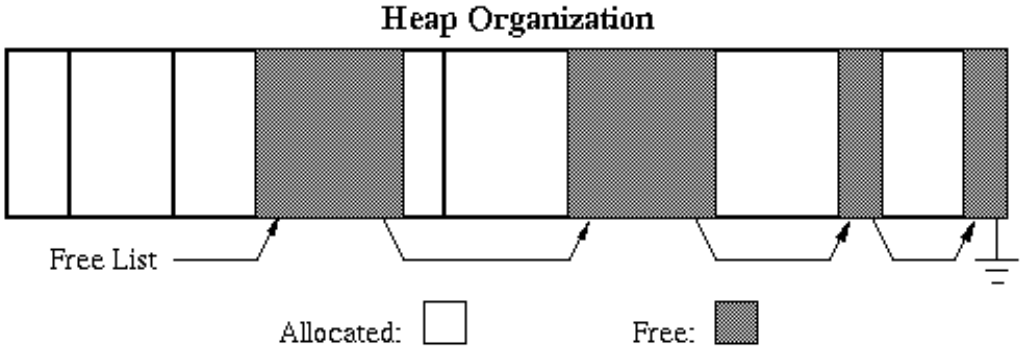
Stack organization: memory allocation and freeing are partially predictable (as usual, we do better when we can predict the future). Allocation is hierarchical: memory is freed in opposite order from allocation. If alloc(A) then alloc(B) then alloc(C), then it must be free(C) then free(B) then free(A).

- Example: procedure call. Program calls Y, which calls X. Each call pushes another stack frame on top of the stack. Each stack frame has space for variable, parameters, and return addresses.
- Stacks are also useful for lots of other things: tree traversal, expression evaluation, top-down recursive descent parsers, etc.

A stack-based organization keeps all the free space together in one place.



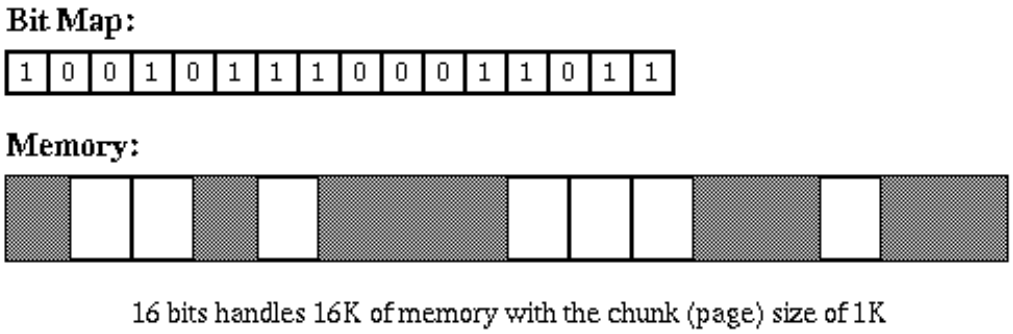
Heap organization: allocation and release are unpredictable. Heaps are used for arbitrary list structures, complex data organizations. Example: payroll system. Do not know when employees will join and leave the company, must be able to keep track of all them using the least possible amount of storage.



- Inevitably end up with lots of holes. Goal: reuse the space in holes to keep the number of holes small, their size large.
- Fragmentation: inefficient use of memory due to holes that are too small to be useful. In stack allocation, all the holes are together in one big chunk.
- Refer to Knuth volume 1 for detailed treatment of what follows.
- Typically, heap allocation schemes use a free list to keep track of the storage that is not in use. Algorithms differ in how they manage the free list.

    - Best fit: keep linked list of free blocks, search the whole list on each allocation, choose block that comes closest to matching the needs of the allocation, save the excess for later. During release operations, merge adjacent free blocks.
    - First fit: just scan list for the first hole that is large enough. Free excess. Also merge on releases. Most first fit implementations are rotating first fit.

- Bit Map: used for allocation of storage that comes in fixed-size chunks (e.g. disk blocks, or 32-byte chunks). Keep a large array of bits, one for each chunk. If bit is 0 it means chunk is in use, if bit is 1 it means chunk is free. Will be discussed more when talking about file systems.



16 bits handles 16K of memory with the chunk (page) size of 1K

Pools: keep a separate allocation pool for each popular size. Allocation is fast, no fragmentation.

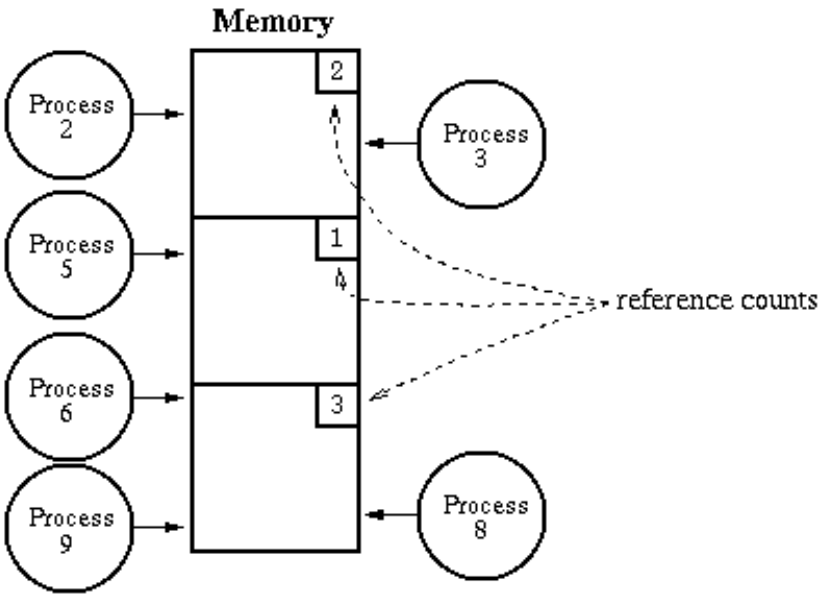
Reclamation Methods: how do we know when memory can be freed?

- It is easy when a chunk is only used in one place.
- Reclamation is hard when information is shared: it cannot be recycled until all of the sharers are finished. Sharing is indicated by the presence of pointers to the data (show example). Without a pointer, cannot access (cannot find it).

Two problems in reclamation:

- Dangling pointers: better not recycle storage while it is still being used.
- Core leaks: Better not "lose" storage by forgetting to free it even when it cannot ever be used again.

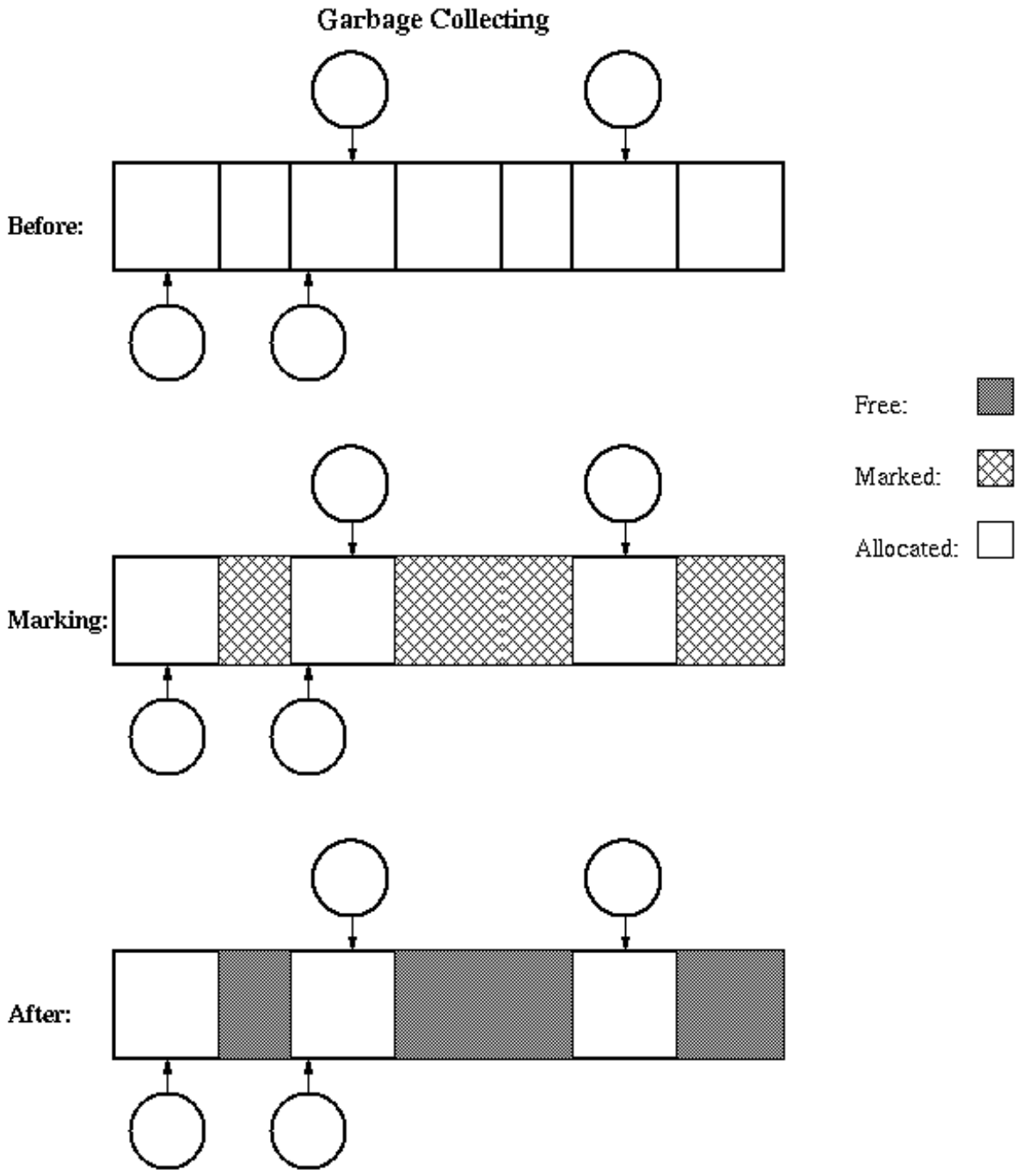
Reference Counts: keep track of the number of outstanding pointers to each chunk of memory. When this goes to zero, free the memory. Example: Smalltalk, file descriptors in Unix. Works fine for hierarchical structures. The reference counts must be managed automatically (by the system) so no mistakes are made in incrementing and decrementing them.



Garbage Collection: storage is not freed explicitly (using free operation), but rather implicitly: just delete pointers. When the system needs storage, it searches through all of the pointers (must be able to find them all!) and collects things that are not used. If structures are circular then this is the only way to reclaim space. Makes life easier on the application programmer, but garbage collectors are incredibly difficult to program and debug, especially if compaction is also done. Examples: Lisp, capability systems.

How does garbage collection work?

- Must be able to find all objects.
- Must be able to find all pointers to objects.
- Pass 1: mark. Go through all pointers that are known to be in use: local variables, global variables. Mark each object pointed to, and recursively mark all objects it points to.
- Pass 2: sweep. Go through all objects, free up those that are not marked.



Garbage collection is often expensive: 20% or more of all CPU time in systems that use it.

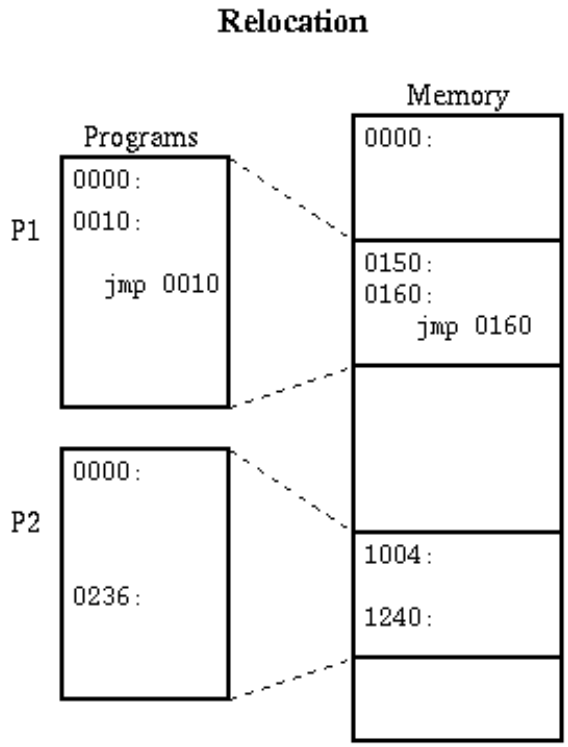
Sharing Main Memory

Issues:

- Want to let several processes coexist in main memory.
- No process should need to be aware of the fact that memory is shared. Each must run regardless of the number and/or locations of processes.
- Processes must not be able to corrupt each other.
- Efficiency (both of CPU and memory) should not be degraded badly by sharing. After all, the purpose of sharing is to increase overall efficiency.

Relocation: draw a simple picture of memory with some processes in it.

- Because several processes share memory, we cannot predict in advance where a process will be loaded in memory. This is similar to a compiler's inability to predict where a subroutine will be after linking.



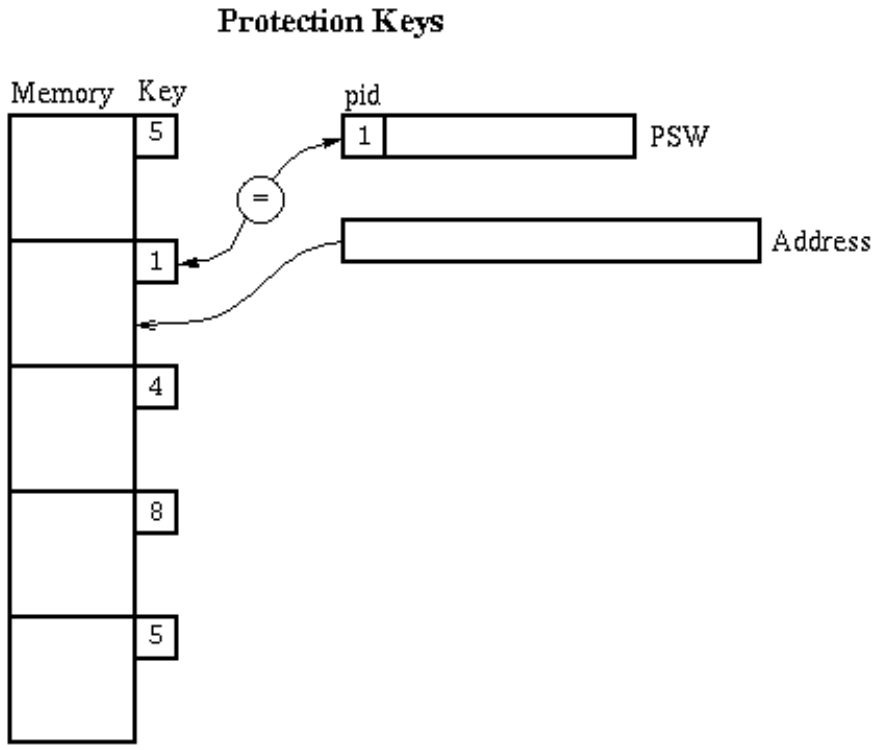
- Relocation adjusts a program to run in a different area of memory. Linker is an example of static relocation used to combine modules into

programs. We now look at relocation techniques that allow several programs to share one main memory.

Static software relocation, no protection:

- Lowest memory holds OS.
- Processes are allocated memory above the OS.
- When a process is loaded, relocate it so that it can run in its allocated memory area (just like linker: linker combines several modules into one program, OS loader combines several processes to fit into one memory; only difference is that there are no cross-references between processes).
- Problem: any process can destroy any other process and/or the operating system.
- Examples: early batch monitors where only one job ran at a time and all it could do was wreck the OS, which would be rebooted by an operator. Many of today's personal computers also operate in a similar fashion.

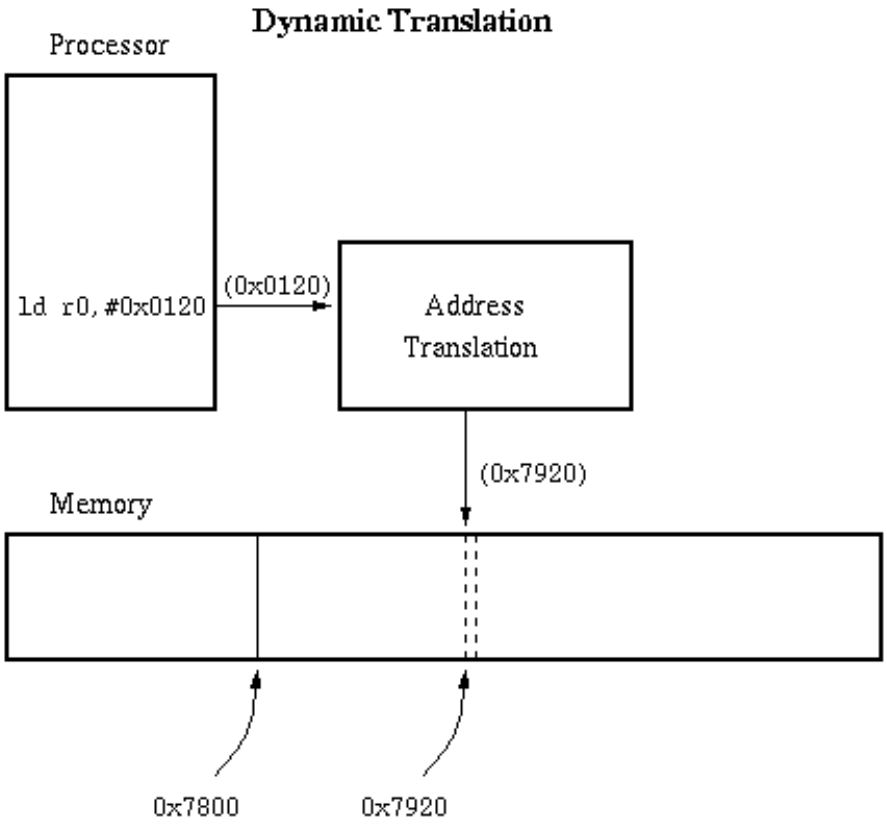
Static relocation with protection keys (IBM S/360 approach):



- Protection Key = a small integer stored with each chunk of memory. The chunks are likely to be 1k-4k bytes.
- Keep an extra hardware register to identify the current process. This is called the process id, or PID. 0 is reserved for the operating system's process id.
- On every memory reference, check the PID of the current process against the key of the memory chunk being accessed. PID 0 is allowed to touch anything, but any other mismatch results in an error trap.
- Additional control: who is allowed to set the PID? How does OS regain control once it has given it up?
- This is the scheme used for the IBM S/360 family. It is safe but inconvenient:
    - Programs have to be relocated before loading. In some systems (e.g. MPS) this requires complete relinking. Expensive.
    - Cannot share information between two processes very easily
    - Cannot swap a process out to secondary storage and bring it back to a different location

Dynamic memory relocation: instead of changing the addresses of a program before it is loaded, we change the address dynamically during every reference.

- Under dynamic relocation, each program-generated address (called a logical or virtual address) is translated in hardware to a physical, or real address. This happens as part of each memory reference.

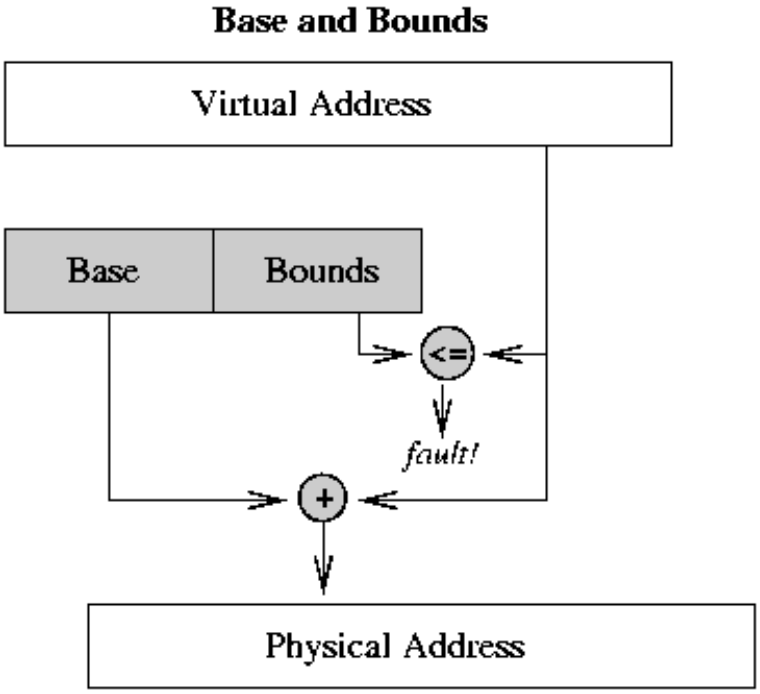


- Show how dynamic relocation leads to two views of memory, called address spaces. With static relocation we force the views to coincide so that there can be several levels of mapping.

Base and Bounds, Segmentation

Base & bounds relocation:

- Two hardware registers: base address for process, bounds register that indicates the last valid address the process may generate.



Each process must be allocated contiguously in real memory.

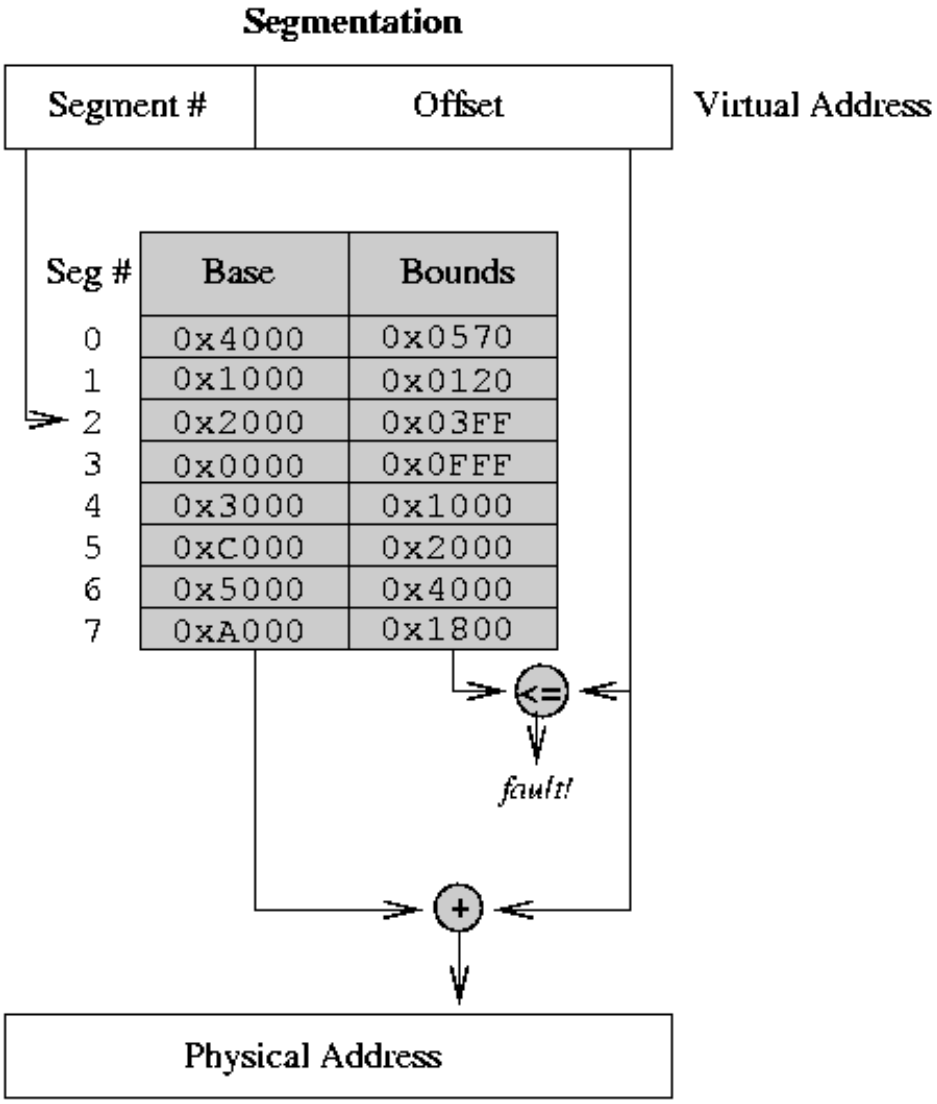
- On each memory reference, the virtual address is compared to the bounds register, then added to the base register. A bounds violation results in an error trap.
- Each process appears to have a completely private memory of size equal to the bounds register plus 1. Processes are protected from each other. No address relocation is necessary when a process is loaded.
- Typically, the OS runs with relocation turned off, and there are special instructions to branch to and from the OS while at the same time turning relocation on and off. Modification of the base and bounds registers must also be controlled.
- Base & bounds is cheap -- only 2 registers -- and fast -- the add and compare can be done in parallel.
- Explain how swapping can work.
- Examples: CRAY-1.

Problem with base&bound relocation:

- Only one segment. How can two processes share code while keeping private data areas (e.g. shared editor)? Draw a picture to show that it cannot be done safely with a single-segment scheme.

Multiple segments.

- Permit process to be split between several areas of memory. Each area is called a segment and contains a collection of logically-related information, e.g. code or data for a module.



- Use a separate base and bound for each segment, and also add a protection bit (read/write).
- Each memory reference indicates a segment and offset in one or more of three ways:
    - Top bits of address select segment, low bits the offset. This is the most common, and the best.

    - Or, segment is selected implicitly by the operation being performed (e.g. code vs. data, stack vs. data).
    - Or, segment is selected by fields in the instruction (as in Intel x86 prefixes).

(The last two alternatives are kludges used for machines with such small addresses that there is not room for both a segment number and an offset)

Segment table holds the bases and bounds for all the segments of a process.

Show memory mapping procedure, involving table lookup + add + compare. Example: PDP-10 with high and low segments selected by high-order address bit.

Segmentation example: 8-bit segment number, 16-bit offset.

- Segment table (use above picture -- all numbers in hexadecimal):
- Code in segment 0 (addresses are virtual):
- 0x00242:mov 0x60100,%r1
- 0x00246:st %r1,0x30107
- 0x0024A:b 0x20360
- Code in segment 2:
- 0x20360:ld [%r1+2],%r2
- 0x20364:ld [%r2],%r3
- ...
- 0x203C0:ret

Advantage of segmentation: segments can be swapped and assigned to storage independently.

Problems:

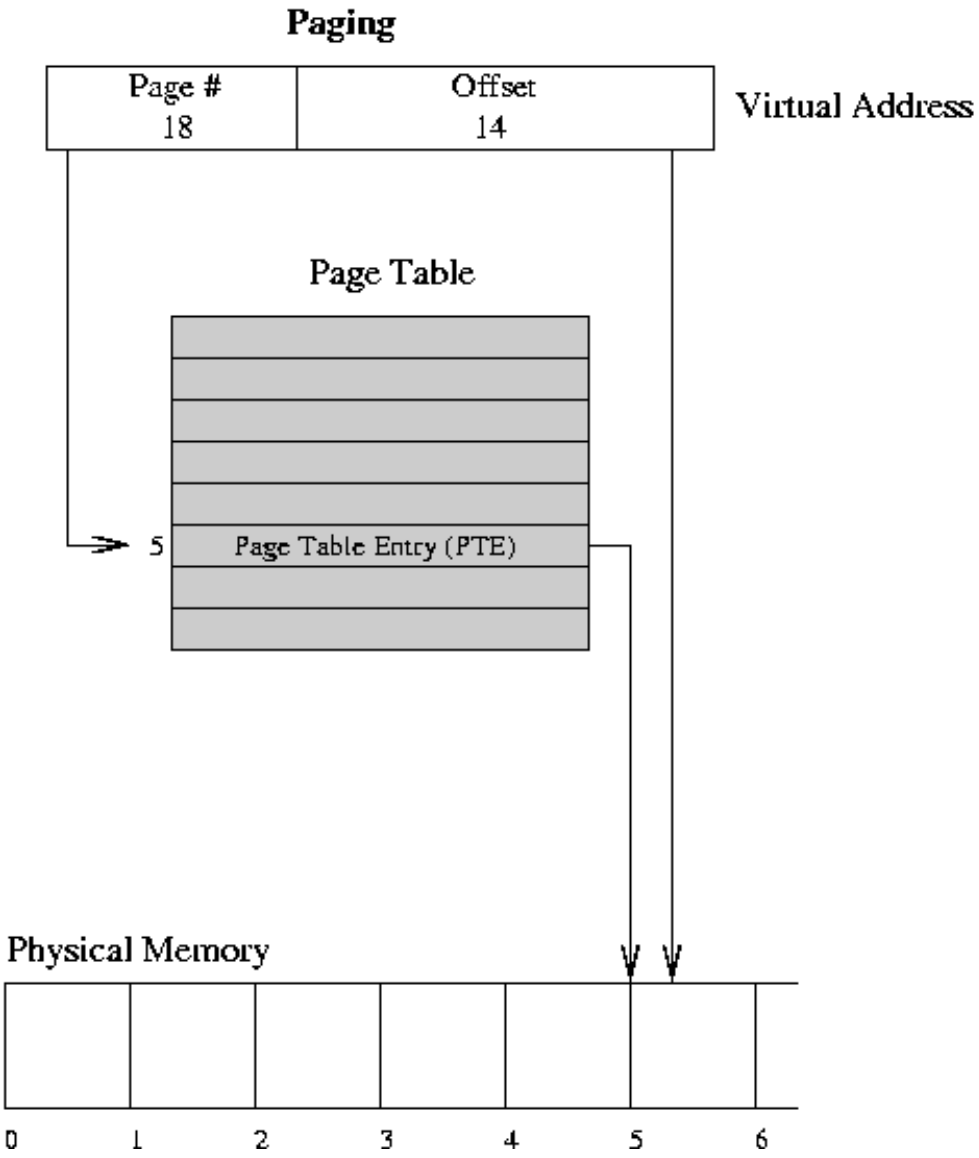
- External fragmentation: segments of many different sizes.
- Segments may be large, have to be allocated contiguously.
- (These problems also apply to base and bound schemes)

Example: in PDP-10's when a segment gets larger, it may have to be shuffled to make room. If things get really bad it may be necessary to compact memory.

Paging

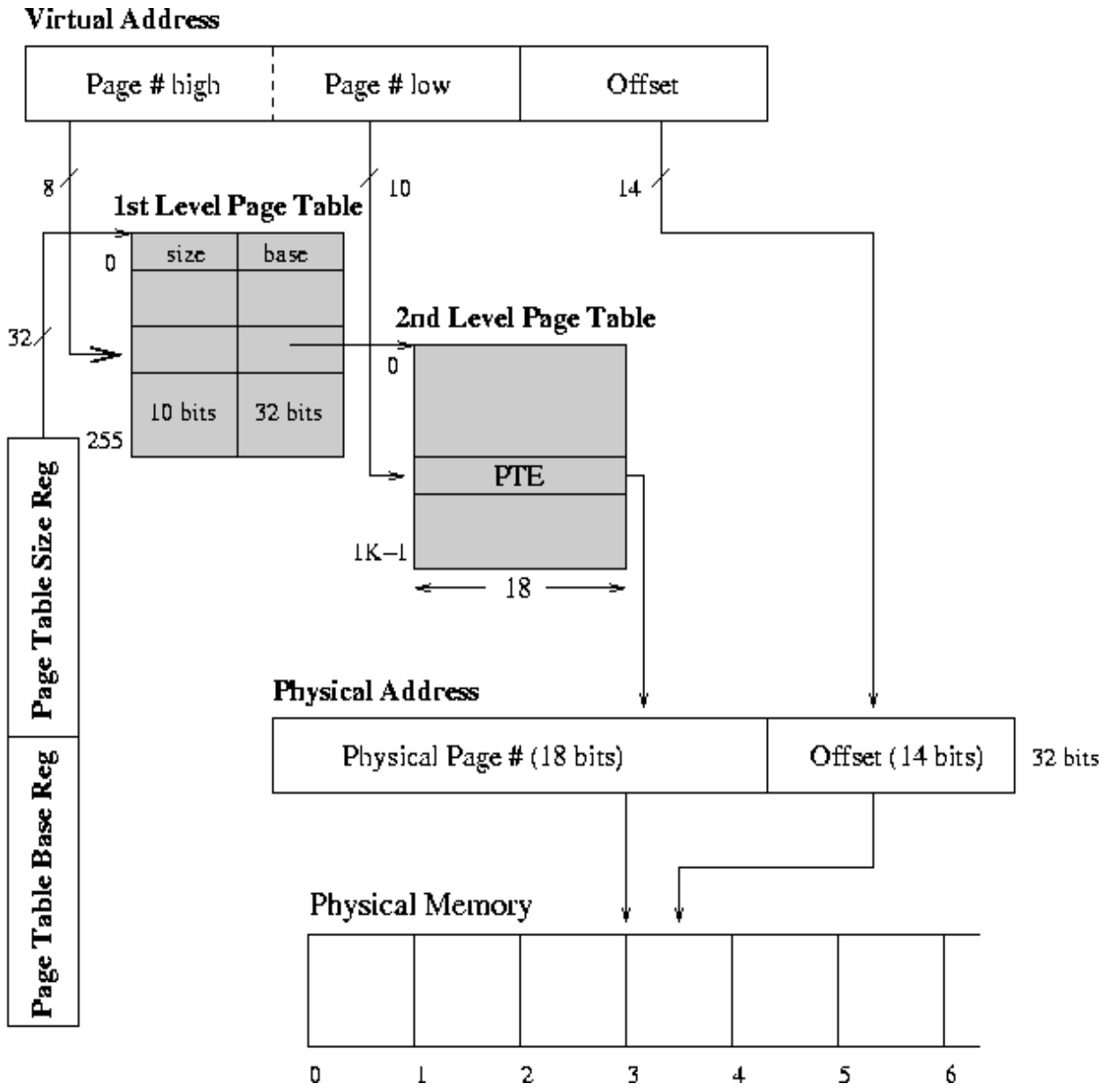
Goal is to make allocation and swapping easier, and to reduce memory fragmentation.

- Make all chunks of memory the same size, call them pages. Typical sizes range from 512-8k bytes.
- For each process, a page table defines the base address of each of that process' pages along with read/only and existence bits.
- Page number always comes directly from the address. Since page size is a power of two, no comparison or addition is necessary. Just do table lookup and bit substitution.
- Easy to allocate: keep a free list of available pages and grab the first one. Easy to swap since everything is the same size, which is usually the same size as disk blocks to and from which pages are swapped.
- Problems:
    - Internal fragmentation: page size does not match up with information size. The larger the page, the worse this is.
    - Table space: if pages are small, the table space could be substantial. In fact, this is a problem even for normal page sizes: consider a 32-bit address space with 1k pages. What if the whole table has to be present at once? Partial solution: keep base and bounds for page table, so only large processes have to have large tables.
    - Efficiency of access: it may take one overhead reference for every real memory reference (page table is so big it has to be kept in memory).



Two-Level (Multi-Level) Paging

Use two levels of mapping to make tables manageable.

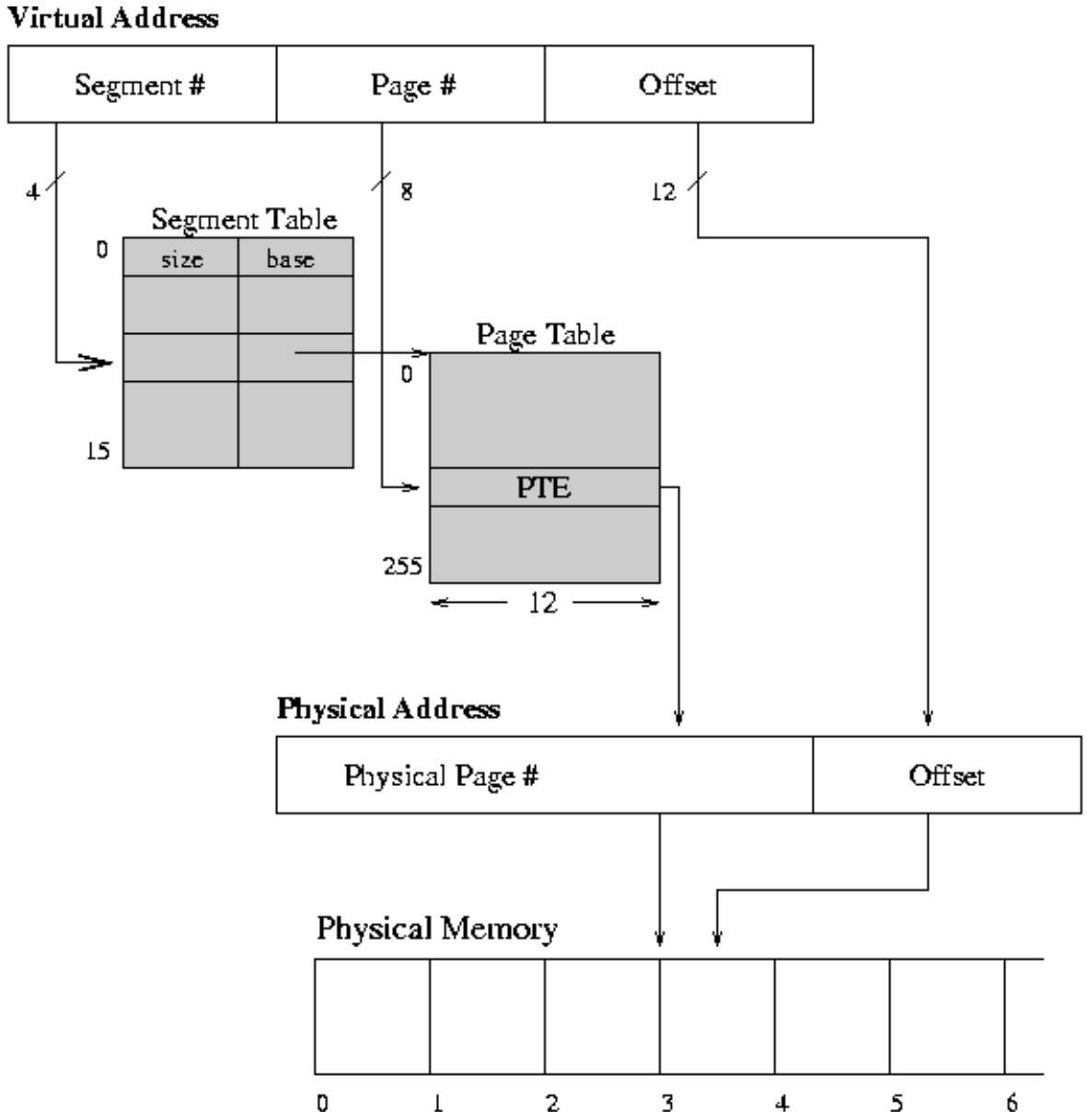


Segmentation and Paging

Use two levels of mapping, with logical sizes for objects, to make tables manageable.

- Each segment contains one or more pages.
- Segment correspond to logical units: code, data, stack. Segments vary in size and are often large. Pages are for the use of the OS; they are fixed size to make it easy to manage memory.

- Going from paging to P+S is like going from single segment to multiple segments, except at a higher level. Instead of having a single page table, have many page tables with a base and bound for each. Call the material associated with each page table a segment.



System 370 example: 24-bit virtual address space, 4 bits of segment number, 8 bits of page number, and 12 bits of offset. Segment table contains real address of page table along with the length of the page table (a sort of bounds register for the segment). Page table entries are only 12 bits, real addresses are 24 bits.

- If a segment is not used, then there is no need to even have a page table for it.
- Can share at two levels: single page, or single segment (whole page table).

Pages eliminate external fragmentation, and make it possible for segments to grow without any reshuffling.

If page size is small compared to most segments, then internal fragmentation is not too bad.

The user is not given access to the paging tables.

If translation tables are kept in main memory, overheads could be very high: 1 or 2 overhead references for every real reference.

Another example: VAX.

- Address is 32 bits, top two select segment. Three base-bound pairs define page tables (system, P0, P1).
- Pages are 512 bytes long.
- Read-write protection information is contained in the page table entries, not in the segment table.
- One segment contains operating system stuff, two contain stuff of current user process.
- Potential problem: page tables can get big. Do not want to have to allocate them contiguously, especially for large user processes. Solution:
    - System base-bounds pairs are physical addresses, system tables must be contiguous.
    - User base-bounds pairs are virtual addresses in the system space. This allows the user page tables to be scattered in non-contiguous pages of physical memory.
    - The result is a two-level scheme.

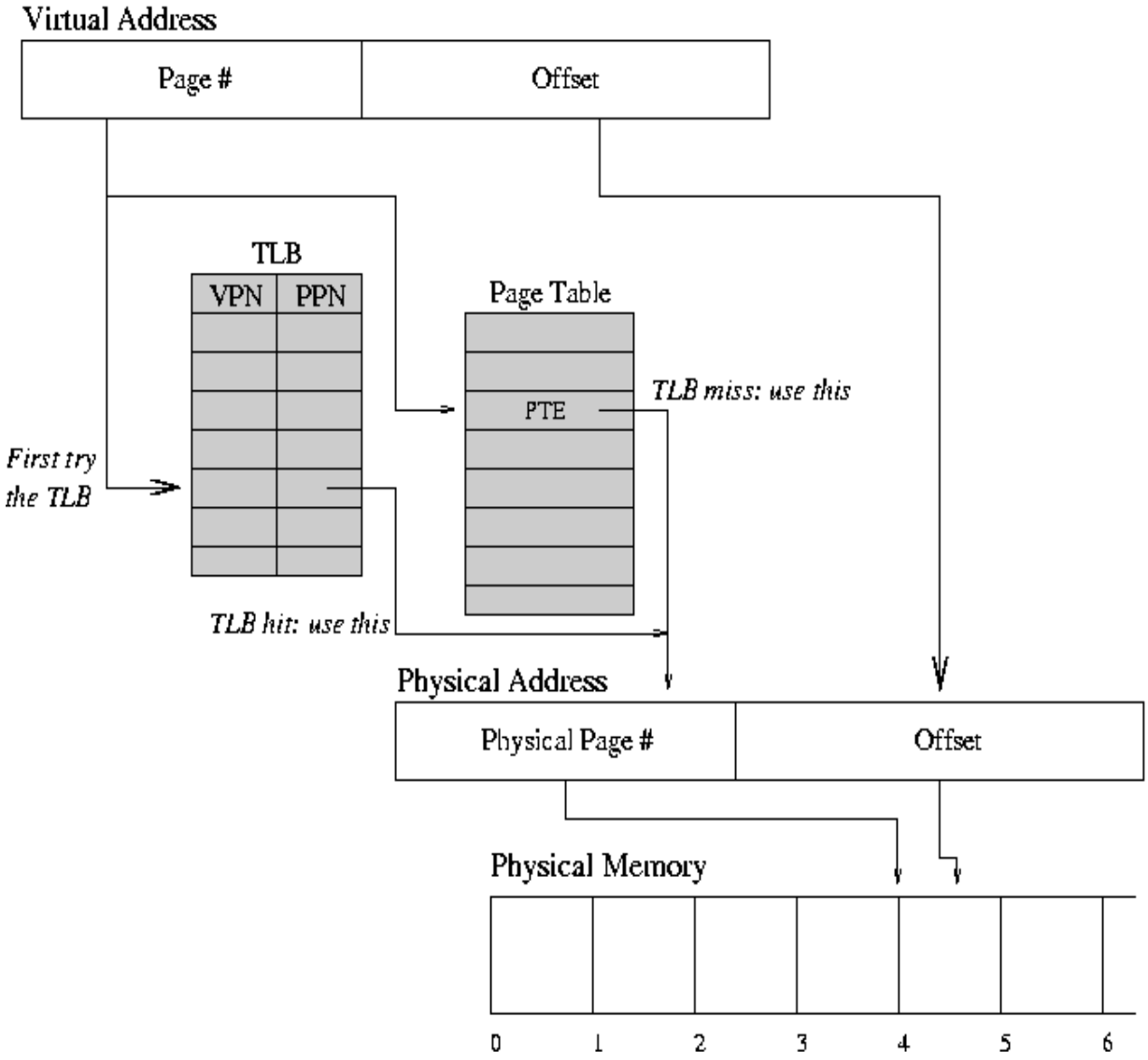
In current systems, you will see three and even four-level schemes to handle 64-bit address spaces.

Translation Buffers and Inverted Page Tables

Problem with segmentation and paging: extra memory references to access translation tables can slow programs down by a factor of two or three. Too many entries in translation tables to keep them all loaded in fast processor memory.

We will re-introduce fundamental concept of locality: at any given time a process is only using a few pages or segments.

Translation Lookaside Buffer



Solution: Translation Lookaside Buffer (TLB). A translation buffer is used to store a few of the translation table entries. It is very fast, but only remembers a small number of entries. On each memory reference:

- First ask TLB if it knows about the page. If so, the reference proceeds fast.
- If TLB has no info for page, must go through page and segment tables to get info. Reference takes a long time, but give the info for this page to TLB so it will know it for next reference (TLB must forget one of its current entries in order to record new one).

TLB Organization: Show picture of black box. Virtual page number goes in, physical page location comes out. Similar to a cache, usually direct mapped.

TLB is just a memory with some comparators. Typical size of memory: 128 entries. Each entry holds a virtual page number and the corresponding physical page number. How can memory be organized to find an entry quickly?

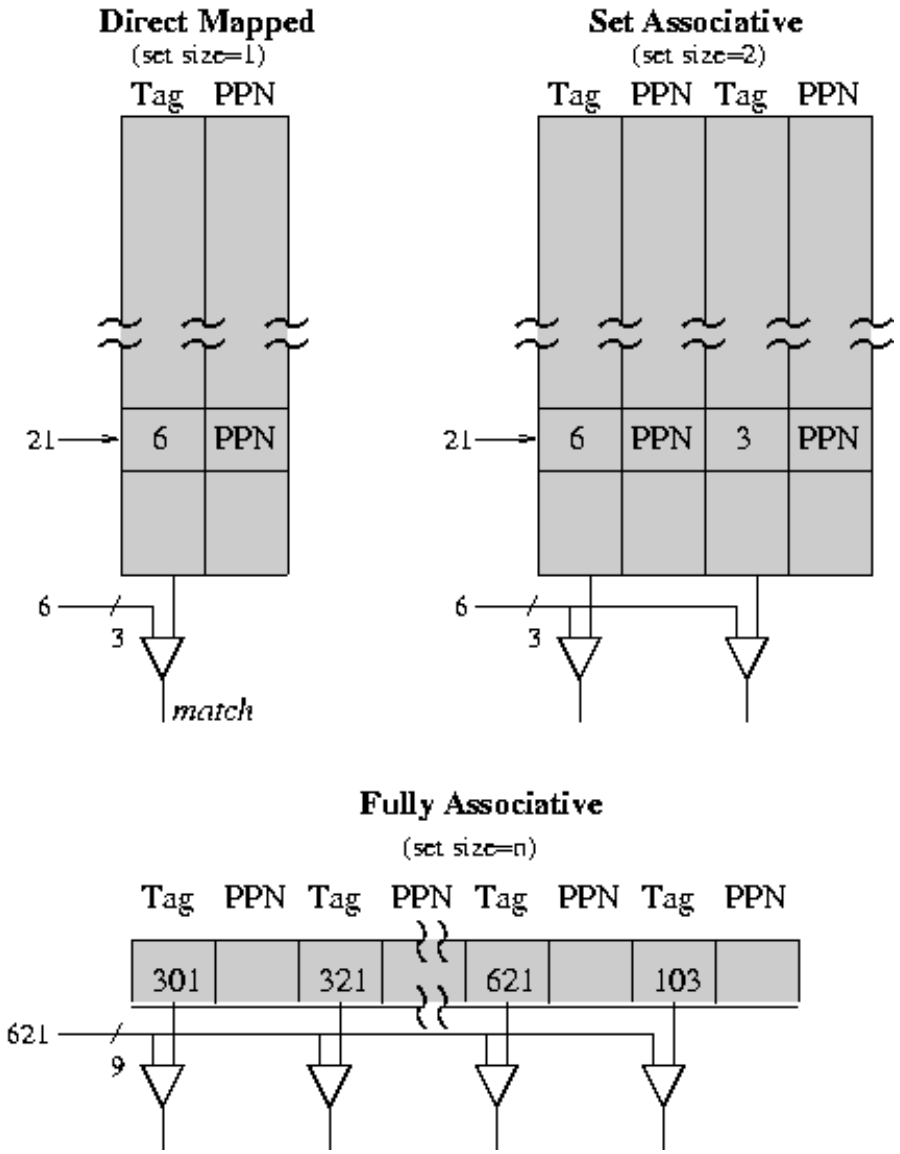
- One possibility: search whole table from start on every reference.
- A better possibility: restrict the info for any given virtual page to fall in exactly one location in the memory. Then only need to check that one location. E.g. use the low-order bits of the virtual page number as the index into the memory. This is the way real TLB's work.

Disadvantage of TLB scheme: if two pages use the same entry of the memory, only one of them can be remembered at once. If process is referencing both pages at same time, TLB does not work very well.

Example: TLB with 64 (100 octal) slots. Suppose the following virtual pages are referenced (octal): 621, 2145, 621, 2145, ... 321, 2145, 321, 621.

TLBs are a lot like hash tables except simpler (must be to be implemented in hardware). Some hash functions are better than others.

- Is it better to use low page number bits than high ones?
- Is there any way to improve on the TLB hashing function?



Another approach: let any given virtual page use either of two slots in the TLB. Make memory wider, use two comparators to check both slots at once.

- This is as fast as the simple scheme, but a bit more expensive (two comparators instead of one, also have to decide which old entry to replace when bringing in a new entry).
- Advantage: less likely that there will be conflicts that degrade performance (takes three pages falling in the same place, instead of two).
- Explain names:

    - Direct mapped.
    - Set associative.
    - Fully associative.

Must be careful to flush TLB during each context swap. Why?

In practice, TLB's have been extremely successful with 95% or great hit rates for relatively small sizes.

Inverted Page Tables

As address spaces have grown to 64 bits, the side of traditional page tables becomes a problem. Even with two-level (or even three or four!) page tables, the tables themselves can become too large.

A solution (used on the IBM Power4 and others) to this problem has two parts:

- A physical page table instead of a logical one. The physical page table is often called an inverted page table. This table contains one entry per page frame. An inverted page table is very good at mapping from physical page to logical page number (as is done by the operating system during a page fault), but not very good at mapping from virtual page number to physical page number (as is done on every memory reference by the processor).
- A TLB fixes the above problem. Since there is no other hardware or registers dedicated to memory mapping, the TLB can be quite a bit larger so that missing-entry faults are rare.

With an inverted page table, most address translations are handled by the TLB. When there is a miss in the TLB, the operating is notified (via an interrupt) and TLB miss-handler is invoked.

Shadow Tables

The operating system can sometimes be thought of as an extension of the abstractions provided by the hardware. However, when the table format is defined by the hardware (such as for a page table entry), you cannot change that format. So, what do you do if you wanted to store additional information, such as last reference time or sharing pointer, in each entry?

The most common solution is a technique that is sometimes called a shadow table. The idea of a shadow is simple (and familiar to Fortran programmers!):

- Consider the hardware defined data structure as an array.
- For the new information that you want to add, define a new (shadow) array.
- There is one entry in the shadow array for each entry in the hardware array.
- For each new item you want to add to the data structure, you add a new data member to the shadow array.

For example, consider the hardware defined page table to be an array of structures:

```
    struct Page_Entry {
        unsigned PageFrame_hi   : 10;  // 42-bit
page frame number
        unsigned PageFrame_mid  : 16;
        unsigned PageFrame_low  : 16;
        unsigned UserRead       :  1;
        unsigned UserWrite      :  1;
        unsigned KernelRead     :  1;
        unsigned KernelWrite    :  1;
        unsigned Reference      :  1;
        unsigned Dirty          :  1;
        unsigned Valid          :  1;
    }

    struct Page_Entry pageTable[TABLESIZE];
```

If you wanted to added a couple of data members, you cannot simply change it to the following:

```
struct Page_Entry {
    unsigned PageFrame_hi   : 10;
    unsigned PageFrame_mid  : 16;
    unsigned PageFrame_low  : 16;
    unsigned UserRead       :  1;
    unsigned UserWrite      :  1;
    unsigned KernelRead     :  1;
    unsigned KernelWrite    :  1;
    unsigned Reference      :  1;
    unsigned Dirty          :  1;
    unsigned Valid          :  1;
    Time_t lastRefTime;
    PageList *shared;
}
```

Instead, you would define a a second array based on this type:

```
struct Page_Entry {                     struct
PE_Shadow {
        unsigned PageFrame_hi   : 10;
Time_t lastRefTime;
        unsigned PageFrame_mid  : 16;
PageList *shared;
        unsigned PageFrame_low  : 16;       }
        unsigned UserRead       :  1;
        unsigned UserWrite      :  1;
        unsigned KernelRead     :  1;
        unsigned KernelWrite    :  1;
        unsigned Reference      :  1;
        unsigned Dirty          :  1;
        unsigned Valid          :  1;
```

```
}

struct Page_Entry pageTable[TABLESIZE];
struct PE_Shadow  pageShadow[TABLESIZE];
```

Virtual Memory, Page Faults

Problem: how does the operating system get information from user memory? E.g. I/O buffers, parameter blocks. Note that the user passes the OS a virtual address.

- In some cases the OS just runs unmapped. Then all it has to do is read the tables and translate user addresses in software. However, addresses that are contiguous in the virtual address space may not be contiguous physically. Thus I/O operations may have to be split up into multiple blocks. Draw an example.
- Suppose the operating system also runs mapped. Then it must generate a page table entry for the user area. Some machines provide special instructions to get at user stuff. Note that under no circumstances should users be given access to mapping tables.
- A few machines, most notably the VAX, make both system information and user information visible at once (but the user cannot touch system stuff unless the program is running with special kernel protection bit set). This makes life easy for the kernel, although it does not solve the I/O problem.

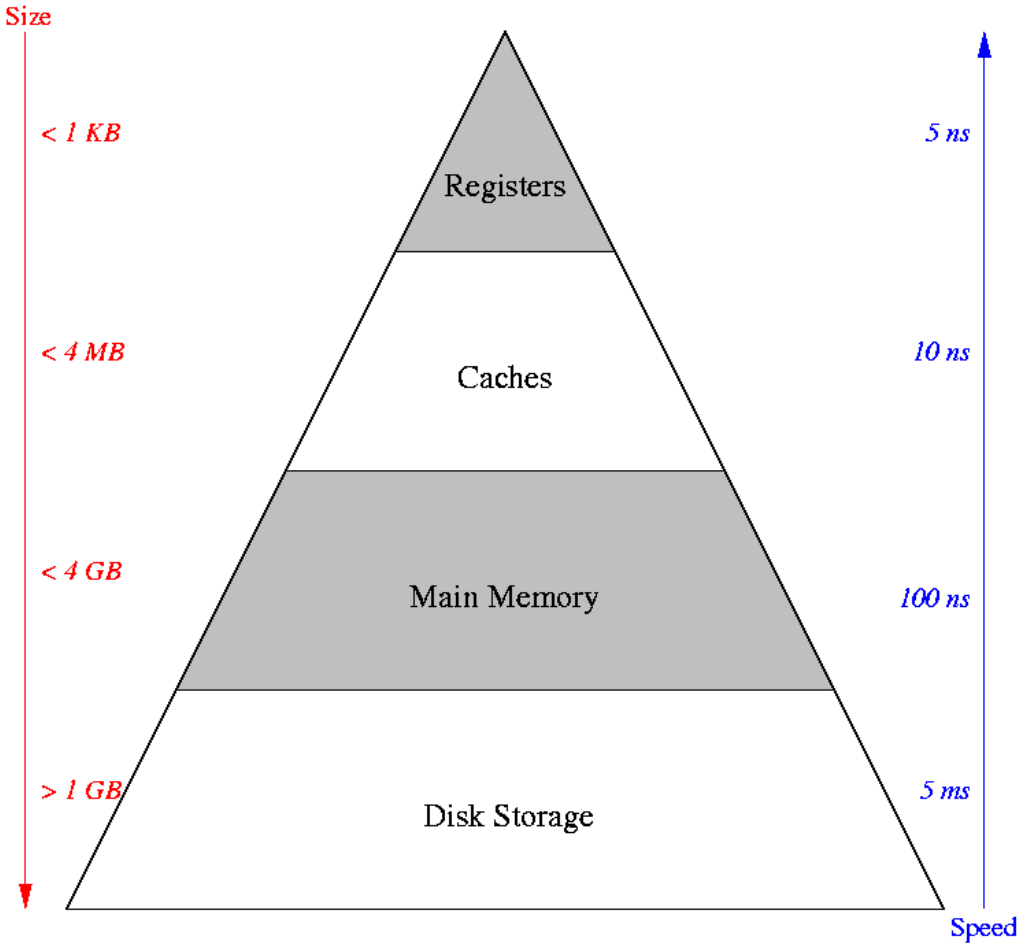
So far we have disentangled the programmer's view of memory from the system's view using a mapping mechanism. Each sees a different organization. This makes it easier for the OS to shuffle users around and simplifies memory sharing between users.

However, until now a user process had to be completely loaded into memory before it could run. This is wasteful since a process only needs a small amount of its total memory at any one time (locality). Virtual memory permits a process to run with only some of its virtual address space loaded into physical memory.

The Memory Hierarchy

The idea is to produce the illusion of a memory with the size of the disk and the speed of main memory.

Data can be in registers (very fast), caches (fast), main memory (not so fast, or disk (slow). Keep the things that you use frequently as close to you (and as fast to access) as possible.

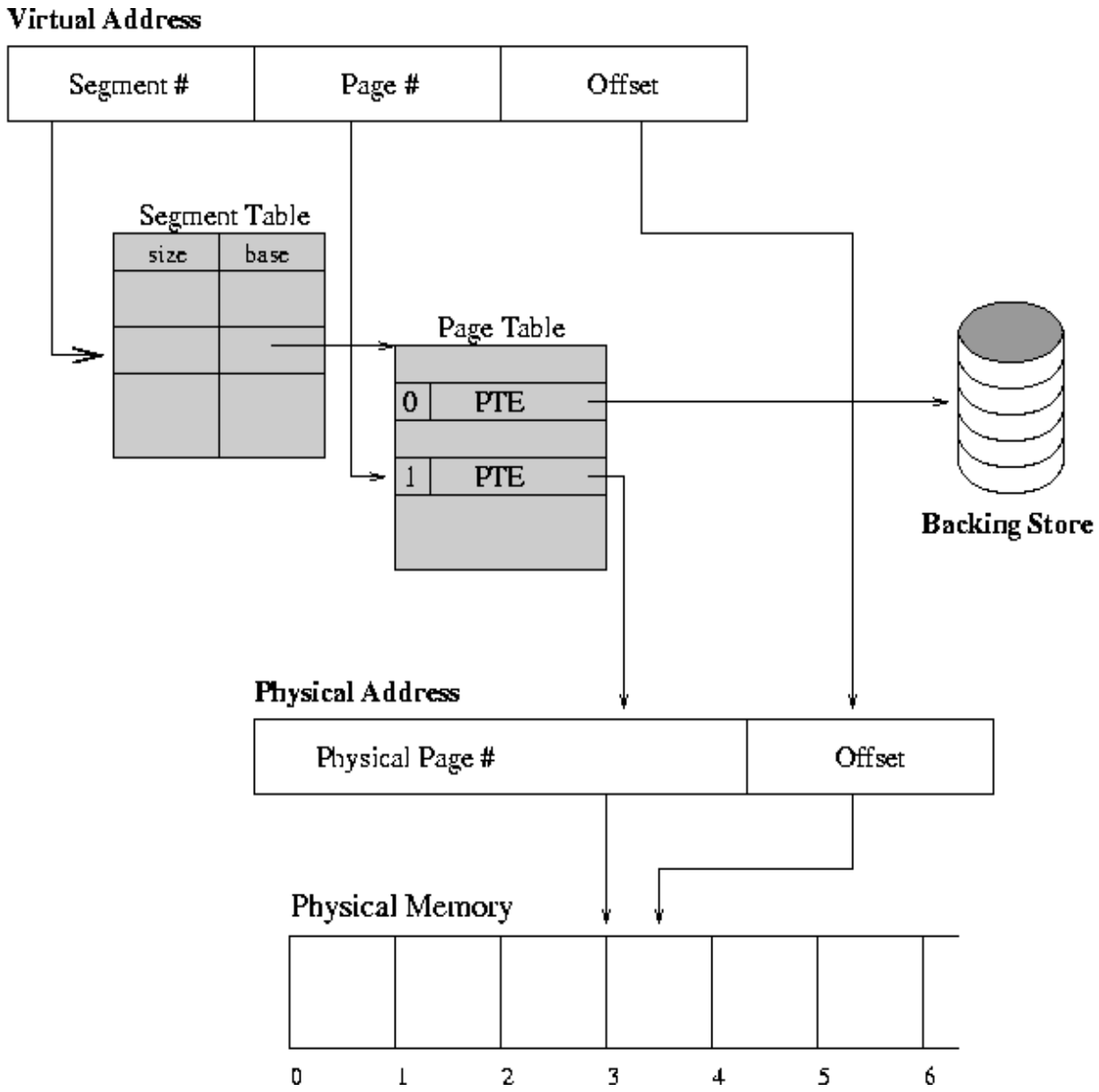


The reason that this works is that most programs spend most of their time in only a small piece of the code. Give Knuth's estimate of 90% of the time in 10% of the code. Introduce again the principle of locality.

Page Faults

If not all of process is loaded when it is running, what happens when it references a byte that is only in the backing store? Hardware and software cooperate to make things work anyway.

- First, extend the page tables with an extra bit "present". If present is not set then a reference to the page results in a trap. This trap is given a special name, page fault.
- Any page not in main memory right now has the "present" bit cleared in its page table entry.
- When page fault occurs:
    - Operating system brings page into memory.
    - Page table is updated, "present" bit is set.
    - The process is continued.



Continuing process is very tricky, since it may have been aborted in the middle of an instruction. Do not want user process to be aware that the page fault even happened.

- Can the instruction just be skipped?
- Suppose the instruction is restarted from the beginning. How is the "beginning" located?
- Even if the beginning is found, what about instructions with side effects, like:

ld [%r2], %r2

- Without additional information from the hardware, it may be impossible to restart a process after a page fault. Machines that permit restarting must have hardware support to keep track of all the side effects so that they can be undone before restarting.
- Forest Baskett's approach for the old Zilog Z8000 (two processors, one just for handling page faults)
- IBM 370 solution (execute long instructions twice).
- If you think about this when designing the instruction set, it is not too hard to make a machine virtualizable. It is much harder to do after the fact. VAX is example of doing it right.

Effective Access Time Calculation

We can calculate the estimated cost of page faults by performing an effective access time calculation. The basic idea is that sometimes you access a location quickly (there is no page fault) and sometimes more slowly (you have to wait for a page to come into memory). We use the cost of each type of access and the percentage of time that it occurs to compute the average time to access a word.

Let:

- h = fraction of time that a reference does not require a page fault.
- tmem = time it takes to read a word from memory.
- tdisk = time it takes to read a page from disk.

then

- EAT = h * tmem + (1 - h) * tdisk.

If there a multiple classes of memory accesses, such as no disk access, one disk access, and two disk access, then you would have a fraction (h) and access time (t) for each class of access.

Note that this calculation is the same type that computer architects use to calculate memory performance. In that case, their access classes might be (1) cached in L1, (2) cached in L2, and (3) RAM.

Page Selection and Replacement

Once the hardware has provided basic capabilities for virtual memory, the OS must make two kinds of scheduling decisions:

- Page selection: when to bring pages into memory.
- Page replacement: which page(s) should be thrown out, and when.

Page selection Algorithms:

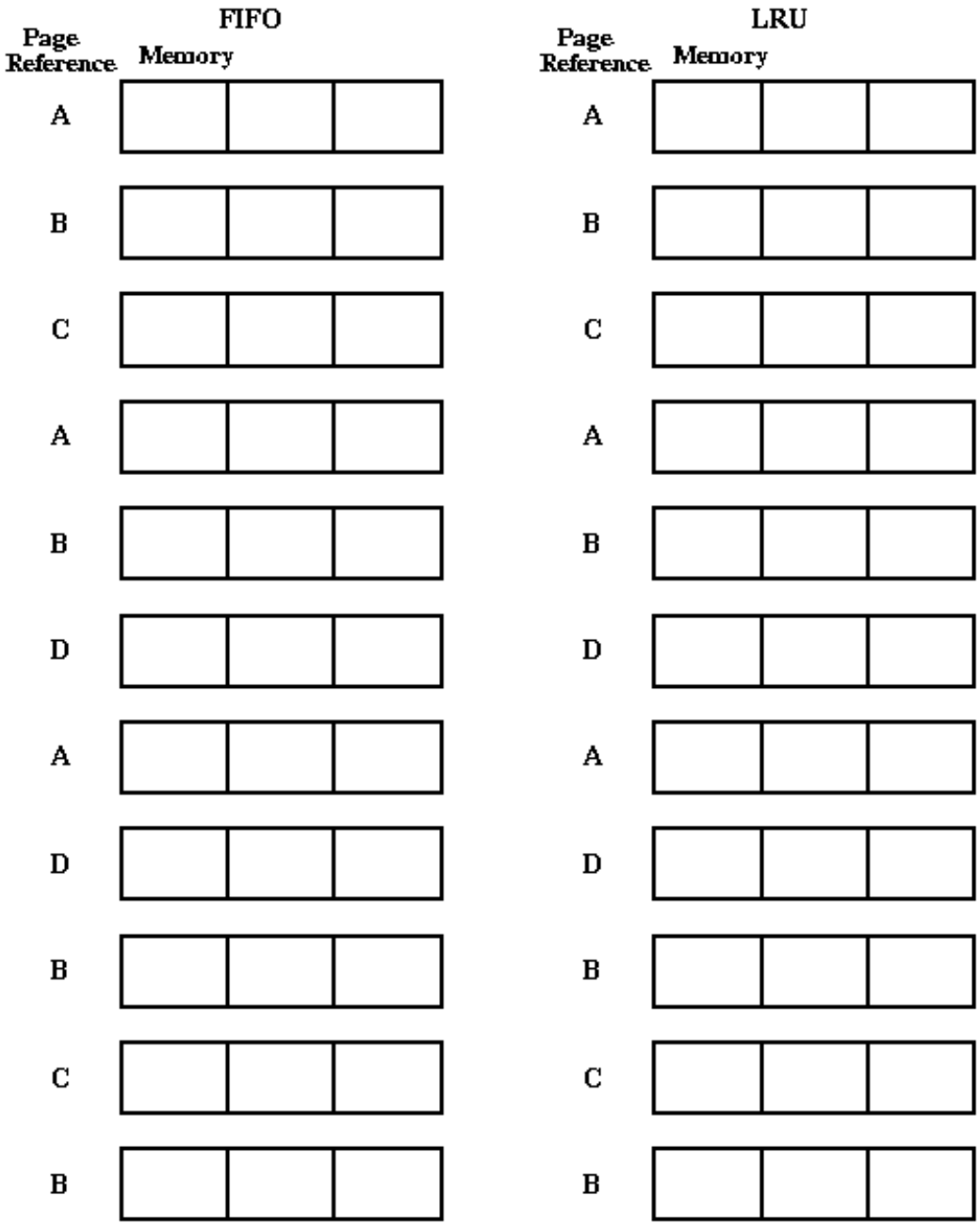
- Demand paging: start up process with no pages loaded, load a page when a page fault for it occurs, i.e. until it absolutely MUST be in memory. Almost all paging systems are like this.
- Request paging: let user say which pages are needed. The trouble is, users do not always know best, and are not always impartial. They will overestimate needs.
- Prepaging: bring a page into memory before it is referenced (e.g. when one page is referenced, bring in the next one, just in case). Hard to do effectively without a prophet, may spend a lot of time doing wasted work.

Page Replacement Algorithms:

- Random: pick any page at random (works surprisingly well!).
- FIFO: throw out the page that has been in memory the longest. The idea is to be fair, give all pages equal residency.
- MIN: naturally, the best algorithm arises if we can predict the future.
- LFU: use the frequency of past references to predict the future.
- LRU: use the order of past references to predict the future.

Example: Try the reference string A B C A B D A D B C B, assume there are three page frames of physical memory. Show the memory allocation

state after each memory reference.



Note that MIN is optimal (cannot be beaten), but that the principle of locality states that past behavior predicts future behavior, thus LRU should do just about as well.

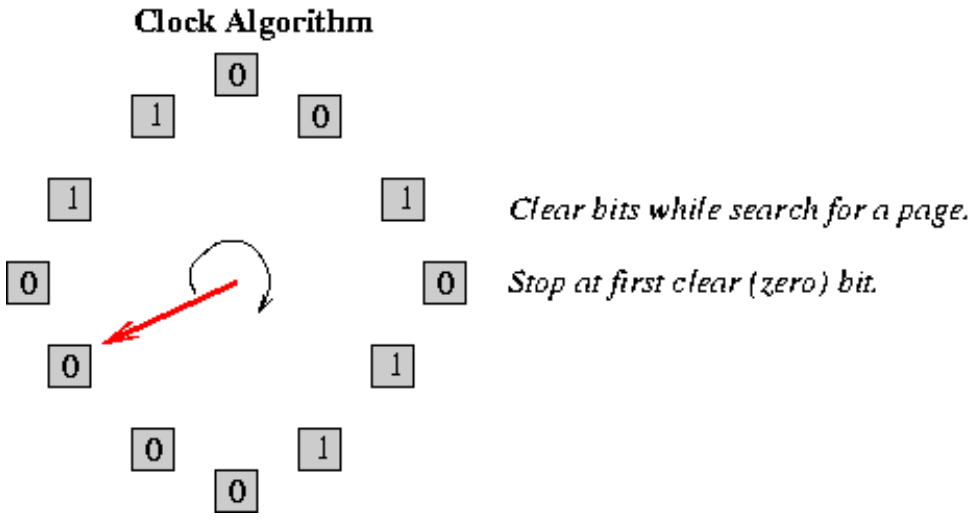
Implementing LRU: need some form of hardware support, in order to keep track of which pages have been used recently.

- Perfect LRU? Keep a register for each page, and store the system clock into that register on each memory reference. To replace a page, scan through all of them to find the one with the oldest clock. This is expensive if there are a lot of memory pages.
- In practice, nobody implements perfect LRU. Instead, we settle for an approximation which is efficient. Just find an old page, not necessarily the oldest. LRU is just an approximation anyway (why not approximate a little more?).

Clock Algorithm, Thrashing

This is an efficient way to approximate LRU.

Clock algorithm: keep "use" bit for each page frame, hardware sets the appropriate bit on every memory reference. The operating system clears the bits from time to time in order to figure out how often pages are being referenced. Introduce clock algorithm where to find a page to throw out the OS circulates through the physical frames clearing use bits until one is found that is zero. Use that one. Show clock analogy.



Fancier algorithm: give pages a second (third? fourth?) chance. Store (in software) a counter for each page frame, and increment the counter if use

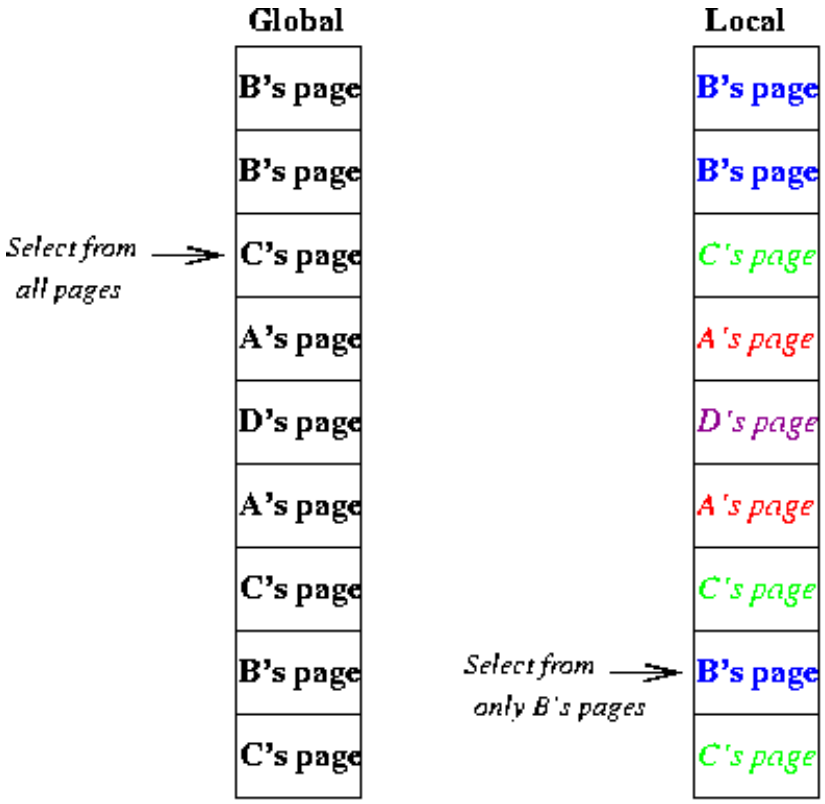
bit is zero. Only throw the page out if the counter passes a certain limit value. Limit = 0 corresponds to the previous case. What happens when limit is small? large?

Some systems also use a "dirty" bit to give preference to dirty pages. This is because it is more expensive to throw out dirty pages: clean ones need not be written to disk.

What does it mean if the clock hand is sweeping very slowly?

What does it mean if the clock hand is sweeping very fast?

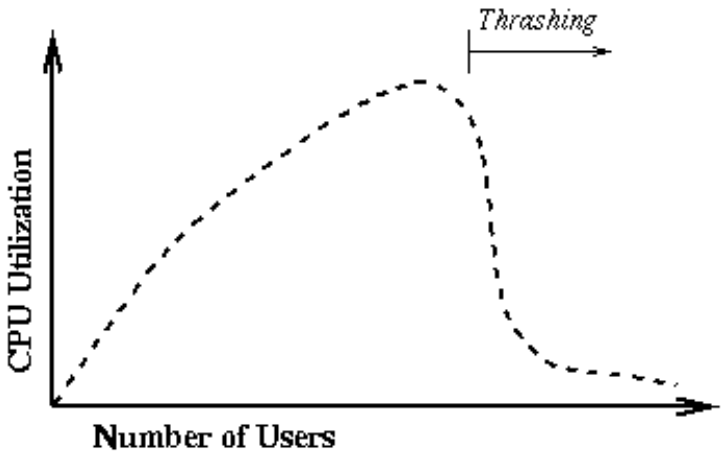
If all pages from all processes are lumped together by the replacement algorithm, then it is said to be a global replacement algorithm. Under this scheme, each process competes with all of the other processes for page frames. A per process replacement algorithm allocates page frames to individual processes: a page fault in one process can only replace one of that process' frames. This relieves interference from other processes. A per job replacement algorithm has a similar effect (e.g. if you run vi it may cause your shell to lose pages, but will not affect other users). In per-process and per-job allocation, the allocations may change, but only slowly.



Thrashing: consider what happens when memory gets overcommitted.

- Suppose there are many users, and that between them their processes are making frequent references to 50 pages, but memory has 40 pages.
- Each time one page is brought in, another page, whose contents will soon be referenced, is thrown out.
- Compute average memory access time.
- The system will spend all of its time reading and writing pages. It will be working very hard but not getting anything done.
- Thrashing was a severe problem in early demand paging systems.

Thrashing occurs because the system does not know when it has taken on more work than it can handle. LRU mechanisms order pages in terms of last access, but do not give absolute numbers indicating pages that must not be thrown out.



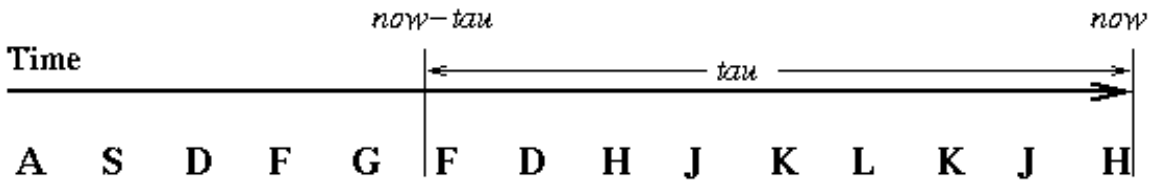
What can be done?

- If a single process is too large for memory, there is nothing the OS can do. That process will simply thrash.
- If the problem arises because of the sum of several processes:
    - Figure out how much memory each process needs.
    - Change scheduling priorities to run processes in groups whose memory needs can be satisfied.

Working Sets

Working Sets are a solution proposed by Peter Denning. An informal definition is "the collection of pages that a process is working with, and which must thus be resident if the process is to avoid thrashing." The idea is to use the recent needs of a process to predict its future needs.

- Choose tau, the working set parameter. At any given time, all pages referenced by a process in its last tau seconds of execution are considered to comprise its working set.



Memory references

- A process will never be executed unless its working set is resident in main memory. Pages outside the working set may be discarded at any time.

Working sets are not enough by themselves to make sure memory does not get overcommitted. We must also introduce the idea of a balance set:

- If the sum of the working sets of all runnable processes is greater than the size of memory, then refuse to run some of the processes (for a while).
- Divide runnable processes up into two groups: active and inactive. When a process is made active its working set is loaded, when it is made inactive its working set is allowed to migrate back to disk. The collection of active processes is called the balance set.
- Some algorithm must be provided for moving processes into and out of the balance set. What happens if the balance set changes too frequently?

As working sets change, corresponding changes will have to be made in the balance set.

Problem with the working set: must constantly be updating working set information.

- One of the initial plans was to store some sort of a capacitor with each memory page. The capacitor would be charged on each reference, then would discharge slowly if the page was not referenced. Tau would be determined by the size of the capacitor. This was not actually implemented. One problem is that we want separate working sets for

each process, so the capacitor should only be allowed to discharge when a particular process executes. What if a page is shared?
- Actual solution: take advantage of use bits
    - OS maintains idle time value for each page: amount of CPU time received by process since last access to page.
    - Every once in a while, scan all pages of a process. For each use bit on, clear page's idle time. For use bit off, add process' CPU time (since last scan) to idle time. Turn all use bits off during scan.
    - Scans happen on order of every few seconds (in Unix, tau is on the order of a minute or more).

Other questions about working sets and memory management in general:

- What should tau be?
    - What if it is too large?
    - What if it is too small?
- What algorithms should be used to determine which processes are in the balance set?
- How do we compute working sets if pages are shared?
- How much memory is needed in order to keep the CPU busy? Note than under working set methods the CPU may occasionally sit idle even though there are runnable processes.

I/O devices and File systems
IO devices and File systems

I/O Device Characteristics:

Terminal.

- One character (8 bits of data or control function) is sent at a time, one interrupt per character.
- 10-1800 characters per second.
- Keyboard and display are independent in most systems (no automatic echo, full duplex).
- Usually handled with one interrupt per character, but sometimes DMA nowadays.

Raster Printers.

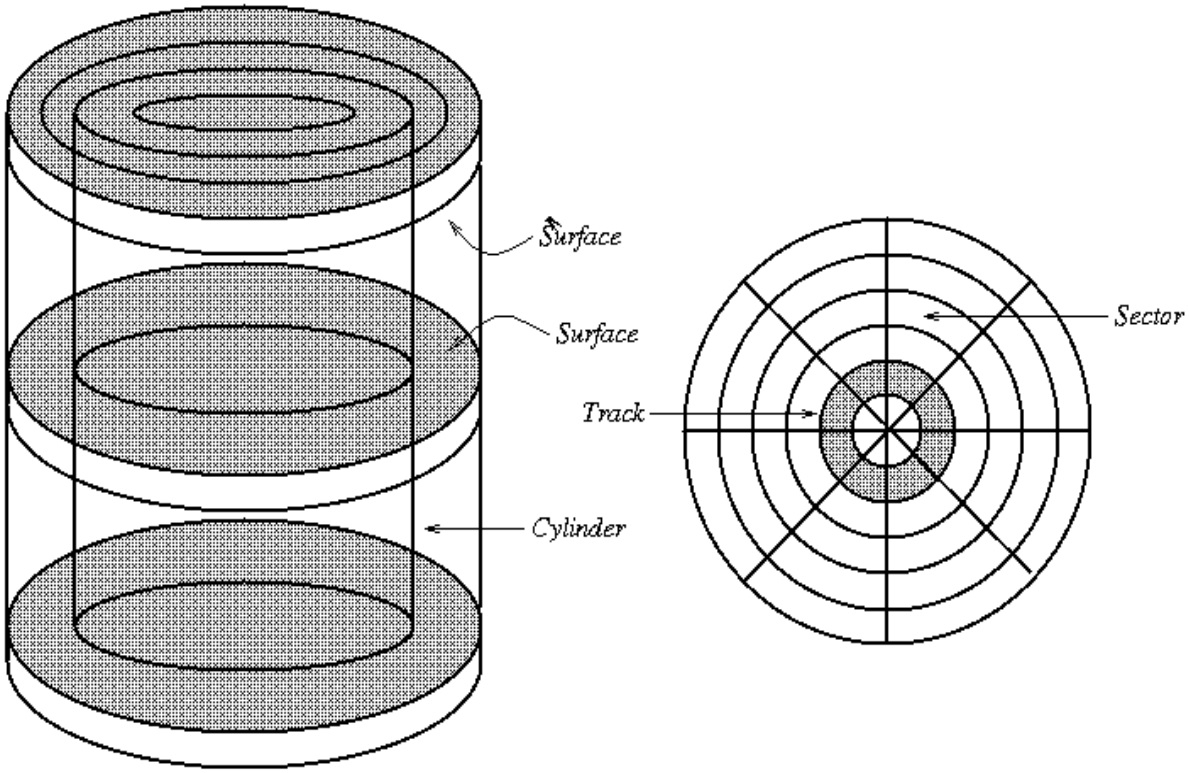
- Bit mapped (one bit for each dot).
- Typically about 300 to 1200 dots per inch.
- Impact, ink jet, or laser printing.

Tape.

- 8mm wide by 750 feet long, or 1/2" wide by 2500 feet long..
- Multi-track, helical, serpentine.
- Variable length records.
- Capacities go up to 70 GB/tape.
- Densities of more than 30 KB/cm.
- Inter-record gaps ofs about 10mm.
- Transfer rates up to 5-500 MB/second.
- Can read or write, but cannot write in middle. Can skip records.
- Tapes are DMA devices, not one interrupt per character.

Disk.

- Draw picture of spindle, platters, read/write-arm.
- Typically about 1024 cylinders, 20 tracks per cylinder, 32 sectors per track, 512 bytes per sector.
- This is a total of about 1 GB per disk, or 500,000 double-spaced typewritten pages. Typically sectors are fixed length (512-4096 bytes are popular sizes).
- Sectors can be read and written individually, or in adjacent groups.
- Seek time = 5-100 ms, latency is 0-15 ms (drive spins at 3600- 7200 RPM). This is the standard for medium term computer storage. Transfer times are about 5-20 MB/second. These times depend on drive, controller, and interface standard (IDE, SCSI).



CD-ROMs.

- Current CD-ROMS's can hold up to 720 MB (650 MB is more typical) of data or 74 minutes of uncompressed audio. The data is organized in one (or a few) continuous spirals. The block size is 2K and "tracks" vary in size from 8 to 23 blocks There is extensive error correction information encoded with the data.
-
- Seek time = 100-300 ms. Transfer times are about .3-1.5 MB/second (for 2X-10X). 1X (150 KB/second) is needed for audio CD's.
- Next generation CD (DVD) should store about 5-10 GB and be fully readable and writable. These drives will handle fully integrated data, audio, and video. There are currently several competing standards (ala the VHS vs. Beta competition of the 80's).

Disks and tapes read and write blocks of information rather than single bytes:

- Storage efficiency: give example for tape. At 1600 bpi, 80-byte records use .05 inch, gaps use .6 inch, tape is all gap. However, 8000-byte records use 5 inches so gaps are only about 11% of the tape. In the case of disks, there are a couple of thousand bits of leader at the beginning of each sector used to identify the sector and to synchronize when reading. For 1000- byte sectors, 20% of the disk space is wasted.
- Access efficiency: on disk it typically takes 25ms overhead before transfer begins. The actual transfer is only about 1 microsecond per byte. Thus one-byte transfers take 25ms total time/byte, 1000-byte transfers take about 25 ms total time/byte. Re-iterate the importance of eliminating seeks.

Two common I/O device access methods: DMA and CPU control

Direct Memory Access (DMA)

In simple systems, the CPU must move each data byte to or from the bus using a LOAD or STORE instruction, as if the data were being moved to memory. This quickly uses up much of the CPU's computational power. In order to allow systems to support high I/O utilization while the CPU is getting useful work done on the users' behalf, devices are allowed to

directly access memory. This direct access of memory by devices (or controllers is called Direct Memory Access, commonly abbreviated DMA).

The CPU is still responsible for scheduling the memory accesses made by DMA devices, but once the program has been established, the CPU has no further involvement until the transfer is complete. Typically DMA devices will issue interrupts on I/O completion.

Because this memory is not being manipulated by the CPU, and therefore addresses may not pass through an MMU, DMA devices often confuse or are confused by virtual memory. It is important to guarantee that memory intended for use by a DMA device is not manipulated by the paging system while the I/O is being performed. Such pages are usually frozen (or pinned) to avoid changes.

In some sense DMA is simply an intermediate step to general purpose programmability on devices and device controllers. Several such smart controllers exist, with features ranging from bit swapping, to digital signal processing, checksum calculations, encryption and compression and general purpose processors. Dealing with that programmability requires synchronization and care. Moreover, in order for code to be portable, writing an interface to such smart peripherals is often a delicate balancing act between making features available and making the device unrecognizable.

I/O Software

The I/O software of the OS has several goals:

- Device Independence: All peripherals performing the same function should have the same interface. All disks should present logical blocks. All network adapters should accept packets. The protection of devices should be managed consistently. For example devices should all be accessible by capability, or all by the file system. In practice this is mitigated by the need to expose some features of the hardware.
- Uniform Naming: The OS needs to have a way to describe the various devices in the system so that it can administer them. Again the naming

system should be as flexible as possible. Systems also have to deal with devices joining or leaving the name space (PCMCIA cards).
- Device Sharing: Most devices are shared at some granularity by processes on a general purpose computer. It's the I/O system's job to make sure that sharing is fair (for some fairness metric) and efficient.
- Error Handling: Devices can often deal with errors without user input - retrying a disk read or something similar. Fatal errors need to be communicated to the user in an understandable manner as well. Furthermore, although hiding errors can be good at some level, at other levels they should be seen. Users must be able to tell that their disks are slowly failing.
- Synchrony and Asynchrony: The I/O system needs to deal with the fact that external devices are not synchronized with the internal clock of the CPU. Events on disk drives occur without any regard for the state of the CPU, and the CPU must deal with that. The I/O system code is what turns the asynchronous interrupts into system events that can be handled by the CPU.

Software Levels

Interrupt Handlers

The Interrupt Service Routines (ISRs) are short routines designed to turn the asynchronous events from devices (and controllers) into synchronous ones that the operating system can deal with in time. While an ISR is executing, some set of interrupts is usually blocked, which is a dangerous state of affairs that should be avoided as much as possible.

ISRs generally encode the information about the interrupt into some queue that the OS checks regularly - e.g. on a context switch.

Device Drivers

Device drivers are primarily responsible for issuing the low-level commands to the hardware that gets the hardware to do what the OS wants.

As a result, many of them are hardware dependent.

Conceptually, perhaps the most important facet of device drivers is the conversion from logical to physical addressing. The OS may be coded in terms of logical block numbers for a file, but it is the device driver that converts such logical addresses to real physical addresses and encodes them in a form that the hardware can understand.

Device drivers may also be responsible for programming smart controllers, multiplexing requests and de-multiplexing responses, and measuring and reporting device performance.

Device Independent OS Code

This is the part of the OS we've really been talking the most about. This part of the OS provides consistent device naming and interfaces to the users. It enforces protection, and does logical level caching and buffering.

In addition to providing a uniform interface, the uniform interface is sometimes pierced at this level to expose specific hardware features -- CD audio capabilities, for instance.

The device independent code also provides a consistent error mode to users, letting them know what general errors occurred when the device driver couldn't recover.

User Code

Even the OS code is relatively rough and ready. User libraries provide simpler interfaces to I/O systems. Good examples are the standard I/O library that provides a simplified interface to the filesystem. printf and fopen are easier to use than write and open. Specifically such systems handle data formatting and buffering.

Beyond that there are user level programs that specifically provide I/O services (daemons). Such programs spool data, or directly provide the services users require.

Files, Disk Management

File: a named collection of bits stored on disk. From the OS' standpoint, the file consists of a bunch of blocks stored on the device. Programmer may actually see a different interface (bytes or records), but this does not matter to the file system (just pack bytes into blocks, unpack them again on reading).

Common addressing patterns:

- Sequential: information is processed in order, one piece after the other. This is by far the most common mode: e.g. editor writes out new file, compiler compiles it, etc.
- Random Access: can address any record in the file directly without passing through its predecessors. E.g. the data set for demand paging, also databases.
- Keyed: search for records with particular values, e.g. hash table, associative database, dictionary. Usually not provided by operating system. TLB is one example of a keyed search.

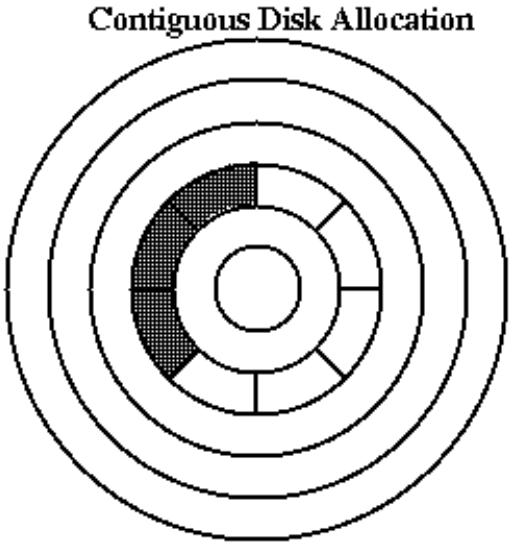
Modern file systems must address four general problems:

- Disk Management: efficient use of disk space, fast access to files, sharing of space between several users.
- Naming: how do users select files?
- Protection: all users are not equal.
- Reliability: information must last safely for long periods of time.

Disk Management: how should the disk sectors be used to represent the blocks of a file? The structure used to describe which sectors represent a file is called the file descriptor.

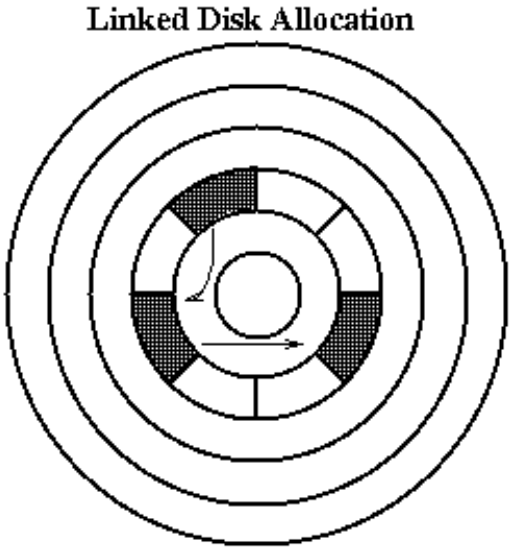
Contiguous allocation: allocate files like segmented memory (give each disk sector a number from 0 up). Keep a free list of unused areas of the

disk. When creating a file, make the user specify its length, allocate all the space at once. Descriptor contains location and size.



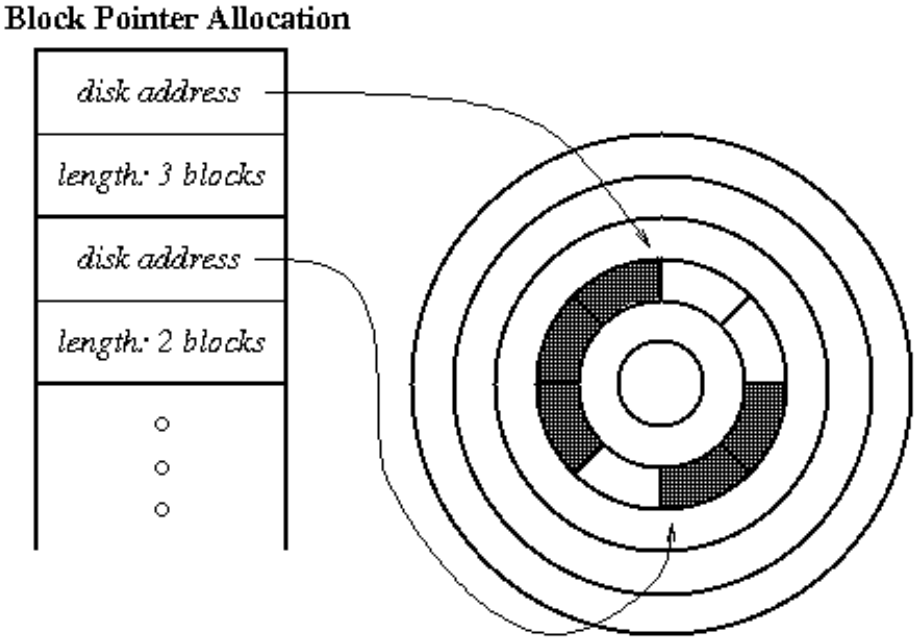
- Advantages: easy access, both sequential and random. Simple. Few seeks.
- Drawbacks: horrible fragmentation will preclude large files, hard to predict needs. With interleaved user requests, still cannot eliminate all seeks.

Linked files: In file descriptor, just keep pointer to first block. In each block of file keep pointer to next block. It can also keep a linked list of free blocks for the free list.



- Advantages: files can be extended, no fragmentation problems. Sequential access is easy: just chase links.
- Drawbacks: random access is virtually impossible. Lots of seeking, even in sequential access.
- Example: FAT (MSDOS) file system.

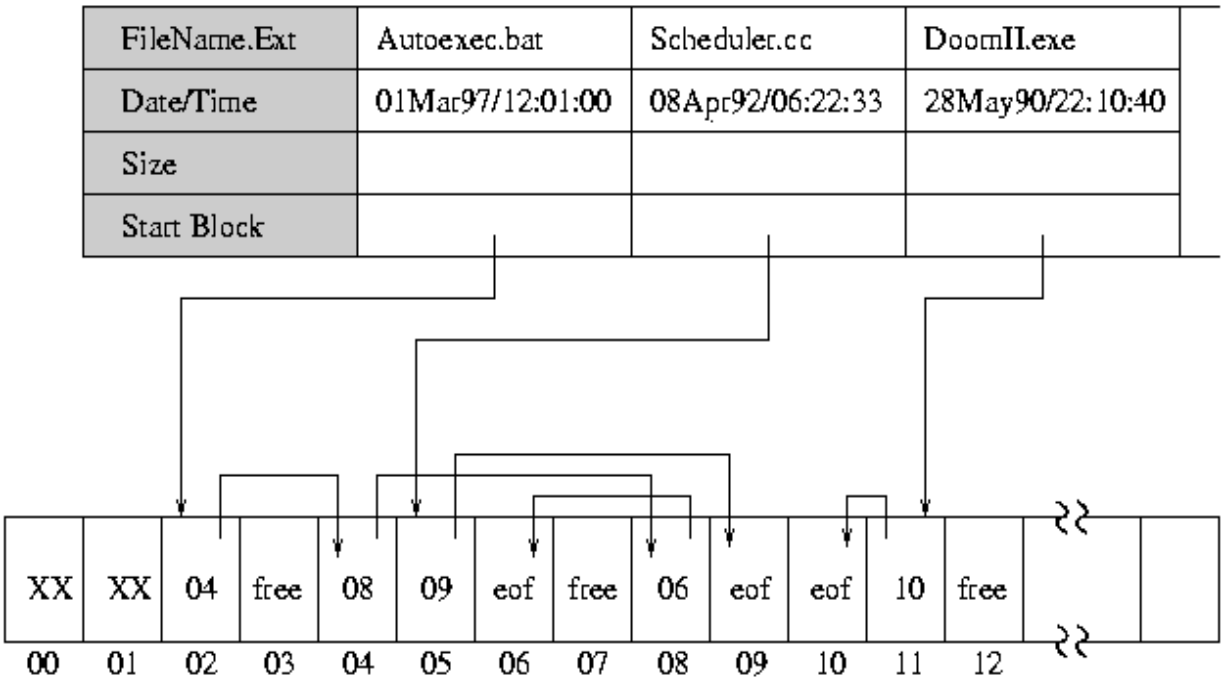
Array of block pointers: file maximum length must be declared when it is created. Allocate an array to hold pointers to all the blocks, but do not allocate the blocks. Then fill in the pointers dynamically using a free list.



- Advantages: not as much space wasted by overpredicting, both sequential and random accesses are easy.
- Drawbacks: still have to set maximum file size, and there will be lots of seeks.

DOS FAT allocation table: A single File Allocation Table (FAT) that combines free list info and file allocation info. In file descriptor, keep pointer to first block. A FAT table entry contains either (1) the block number of the next block in the file, (2) a distinguished "end of file" (eof) value, or (3) a distinguished "free" value.

MS/DOS Directory Entires



File Access Table (FAT)

- Advantages/Disadvantages: similar to those mentioned above for linked file.

None of these is a very good solution: what is the answer? First, and MOST IMPORTANT: understand the application. How are file systems used?

- Most files are small.
- Much of the disk is allocated to large files.
- Many of the I/O's are made to large files.

Thus, the per-file cost must be low, but the performance of large files must be good. File systems must allow reasonably fast random access, extensibility.

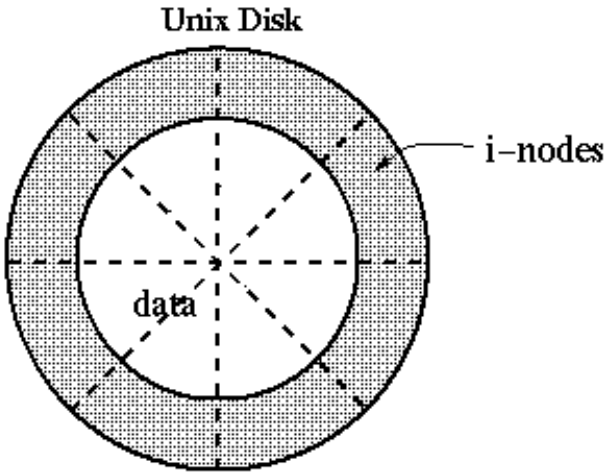
Unix and DEMOS Disk Allocation

Storage Management: For a given file, how the does OS find the blocks contained in that file? The data structure that decribes the contents of file is

generically called a file descriptor. We will see several other names, as we study about file systems.

The file descriptor information has to be stored on disk, so it will stay around even when the OS does not.

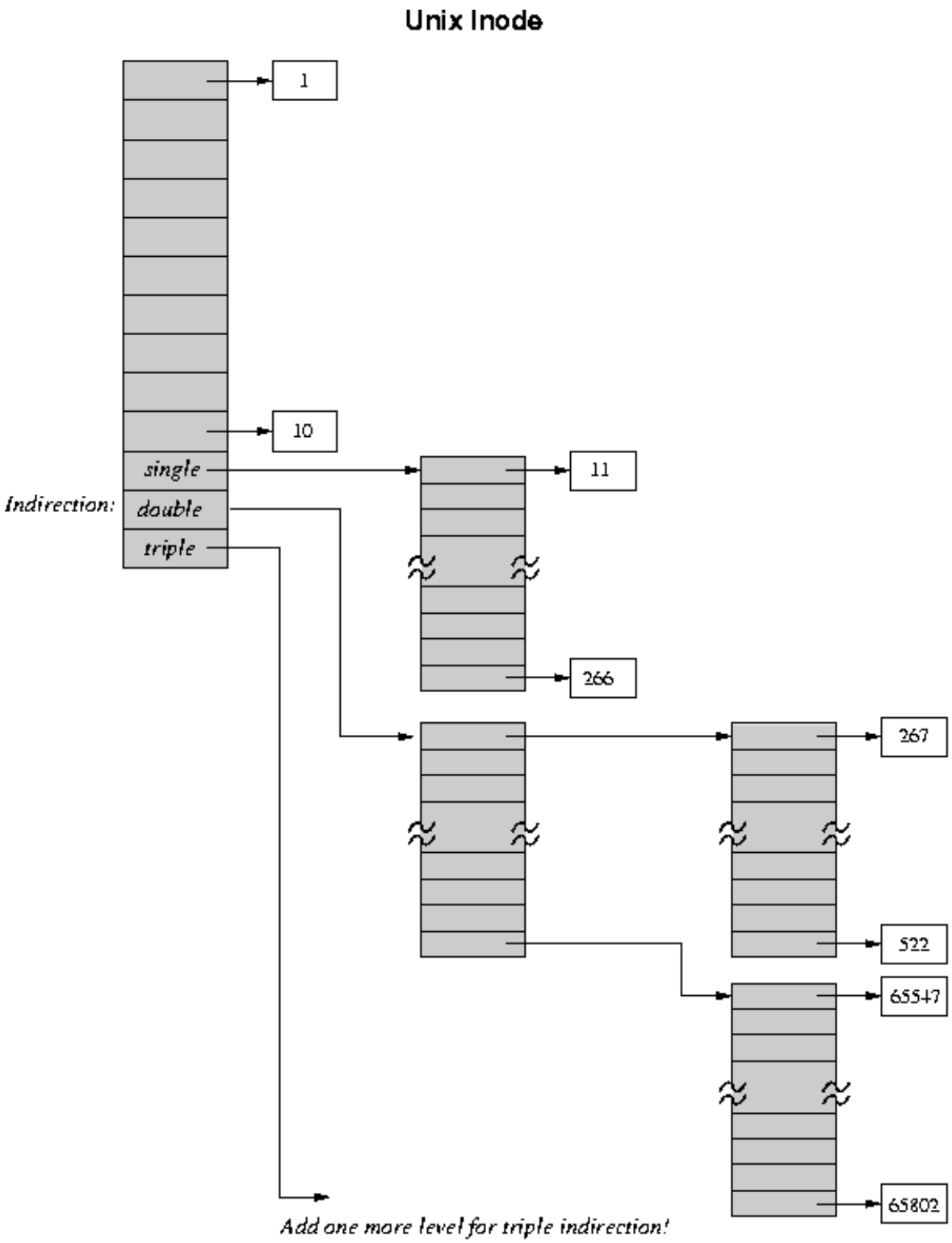
- In Unix, all the descriptors are stored in a fixed size array on disk. The descriptors also contain protection and accounting information.
- A special area of disk is used for this (disk contains two parts: the fixed-size descriptor array, and the remainder, which is allocated for data and indirect blocks).
- The size of the descriptor array is determined when the disk is initialized, and cannot be changed. In Unix, the descriptor is called an i-node, and its index in the array is called its i-number. Internally, the OS uses the i-number to refer to the file.
- When a file is open, its descriptor is kept in main memory. When the file is closed, the descriptor is stored back to disk.



The Typical Unix Inode

- File descriptors: 13 block pointers. The first 10 point to data blocks, the next three to indirect, doubly-indirect, and triply-indirect blocks (256 pointers in each indirect block). Maximum file length is fixed, but large. Descriptor space is not allocated until needed.
- Examples: block 23, block 5 block 340
- Free blocks: stored on a free list in no particular order.

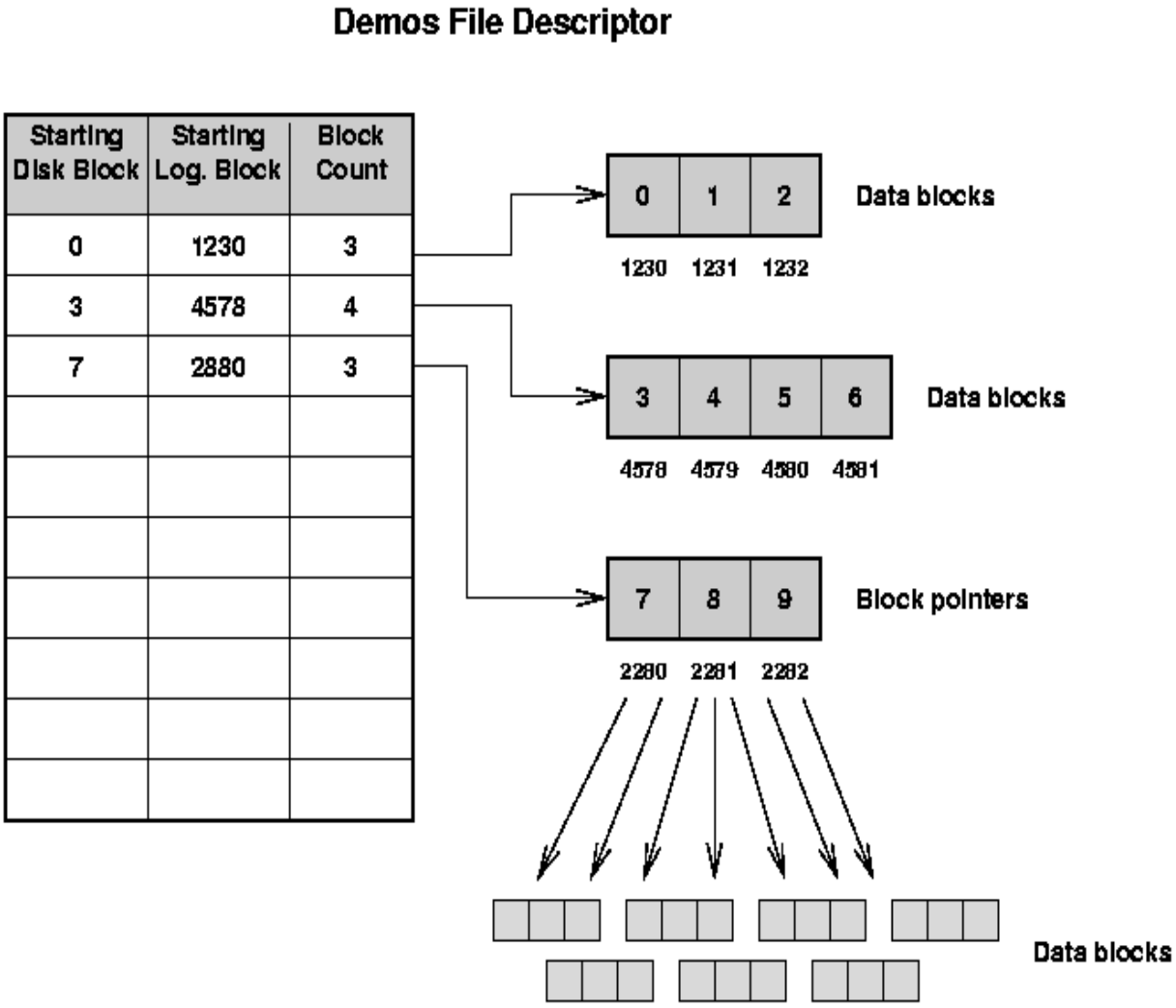
- Go through examples of allocation and freeing.
- Advantages: simple, easy to implement, incremental expansion, easy access to small files.
- Drawbacks:
    - Indirect mechanism does not provide very efficient access to large files: 3 descriptor ops for each real operation. A cache is used, but this takes up main memory space.
    - Block-by-block organization of free list means that that file data gets spread around the disk.



The Demos File System

Demos was an operating system written especially for high performance systems, originally the Cray 1. Its design continues to influence systems today.

The Demos solution: allocates files contiguously, has more compact file descriptors, uses more CPU time. (refer to contiguous allocation picture in section 26).



- File descriptors: select sequences of physical blocks, called block groups, rather than single blocks. Block groups were called extents by IBM.
- A block group has three fields:

    - Starting disk block: the starting address on disk of this block group,
    - Starting logical block: the starting block number within the file for the block group,
    - Count: the number of blocks in the group.

- There are 10 block groups in file descriptor; if files become large, then these become pointers to groups of indirect blocks. The resulting structure is like a B-tree.
- Free blocks: described with a bit map. Just an array of bits, one per block. 1 means block free, 0 means block allocated. For a 300 Mbyte drive there are about 300000 1kbyte blocks, so bit map takes up 40000 bytes. Keep only a small part of the bit map in memory at once. In allocation, scan bit map for adjacent free blocks.
- Advantages:

    - It is easy to allocate block groups, since the bit map automatically merges adjacent free blocks.
    - File descriptors take up less space on disk, require fewer accesses in random access to large files.

- Disadvantages:

    - Slightly more complex than Unix scheme: trades CPU time for disk access time (OK for CRAY-1).
    - When disk becomes full, this becomes VERY expensive, and does not get much in the way of adjacency.

Even if it is possible to allocate in groups, how do you know when to do it? Be guided by history: if file is already big, it will probably get bigger.

Crash Recovery

Computers can crash at any time, and we want the file system to behave sensibly in the face of crashes. The key idea is called consistency:

- The file data and the various control structures (descriptors, bitmaps) must be in agreement.

- Since crashes can occur at any time, not all updates to the disk may be completed.
- We must insure that when the system reboots, it can return its file system to some sensible state.
- The key constraint is that any file system write operation, in progress at the time of the crash, either completely finishes or appears as if it never happened. This is called atomicity by the database folks.

Insuring consistency requires two things:

- Updates to the file system data structures must be done in the write order (and there is only one right order)!
- The proper steps must be taken at reboot time to bring the system back in to a consistent state.

There are three basic updates that happen when data is written to a file.

1. A block (or blocks) is allocated from the free list (bit map).
2. Data is written to the newly allocated block.
3. The inode is updated to include the new data.

These operations must be done in the above order. If they are not, then it is possible to have a data block included in a file that might have garbage (uninitialized data) in the block.

After rebooting, the recovery utility program on Unix, called "fsck", is going to traverse the entire directory structure of the disk to insure that all free blocks are in the free list.

Recovery after a crash follows these steps:

1. Allocate a temporary bit map, initialized to indicate that all disk blocks are free.
2. Start at the inode for the root directory.
3. Traverse the directory:
    - For each disk data block in the directory file, marks its blocks as "allocated" in the bit map.

    - For each data file in this directory, marks its data blocks as "allocated" in the bit map.
    - For each directory in this directory, perform the "Traverse the directory" steps above.

At the completion of the algorithm, you can compare the actual bit map to the temporary one to find blocks that were allocated, but never made it into a file.

Directories

Motivation

Users need a way of finding the files that they created on disk. One approach is just to have users remember descriptor indexes.

Of course, users want to use text names to refer to files. Special disk structures called directories are used to tell what descriptor indices correspond to what names.

A hard concept to understand at the beginning: naming is one of the (if not the) most important issues in systems design.

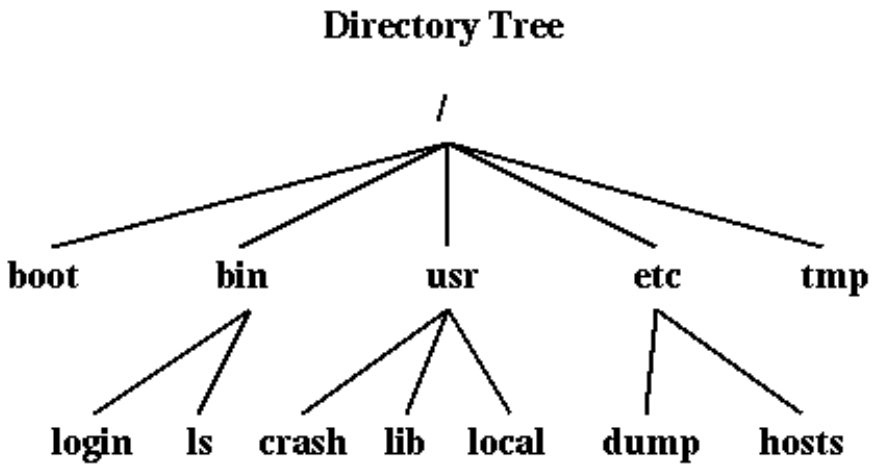
Approach #1: have a single directory for the whole disk. Use a special area of disk to hold the directory.

- Directory contains pairs.
- If one user uses a name, no-one else can.

Approach #2: have a separate directory for each user (TOPS-10 approach). This is still clumsy: names from different projects get confused.

Unix Directories

Unix approach compares the directory structure to a tree.



- Directories are stored on disk just like regular files (i.e. file descriptor with 13 pointers, etc.). User programs can read directories just like any other file (try it!). Only special system programs may write directories.
- Each directory contains pairs. The file pointed to by the index may be another directory. Hence, get hierarchical tree structure, name with /usr/local.
- There is one special directory, called the root. This directory has no name, and is the file pointed to by descriptor 2 (descriptors 0 and 1 have other special purposes).

It is very nice that directories and file descriptors are separate, and the directories are implemented just like files. This simplifies the implementation and management of the structure (can write "normal" programs to manipulate them as files).

Working directory: it is cumbersome constantly to have to specify the full path name for all files.

- In Unix, there is one directory per process, called the working directory, that the system remembers.
- When it gets a file name, it assumes that the file is in the working directory. "/" is an escape to allow full path names.
- Many systems allow more than one current directory. For example, check first in A, then in B, then in C. This set of directories is called the search path or search list. This is very convenient when working on large systems with many different programmers in different areas.

- For example, in Unix the shell will automatically check in several places for programs. However, this is built into the shell, not into Unix, so if any other program wants to do the same, it has to rebuild the facilities from scratch.
- This is yet another example of locality.

Windows (NT) File System

Background

The Windows file system is called NTFS, and was introduced with Windows NT 4.0 and is the standard file system on Windows 2000 and later systems, such as Windows XP. Its goal was to solve the size, performance, reliability, and flexibility limitations in the DOS (aka "FAT" file system).

It has a general similarity to the FAT file system in that all files are described in a single table, called the Master File Table (MFT). However, it has more modern characteristics in that all components are files, including:

FIXME: A LIST CAN NOT BE A TABLE ENTRY. Master File Tabledata filesdirectories	FIXME: A LIST CAN NOT BE A TABLE ENTRY. free list (bit map)boot imagesrecovery logs

The file system also has features to support redundancy and transactions, which we will not discuss. A great reference for details is the book: Inside the Windows NT File System by Helen Custer, published by (not surprisingly) Microsoft Press.

Disk Layout

Disks are divided in fixed size regions:

- Each region is called a volume.
- Each volume can contain a different kind of file system, such as NTFS, FAT, or even Unix.
- Since each volume is a separate file system, it has its own root directory.
- Multiple volumes allow for fixed limits on the growth of a particular file tree, such as limiting the size of temporary file space.
- Multiple volumes also allow a single disk to contain multiple, separating bootable operating systems.

Master File Table (MFT)

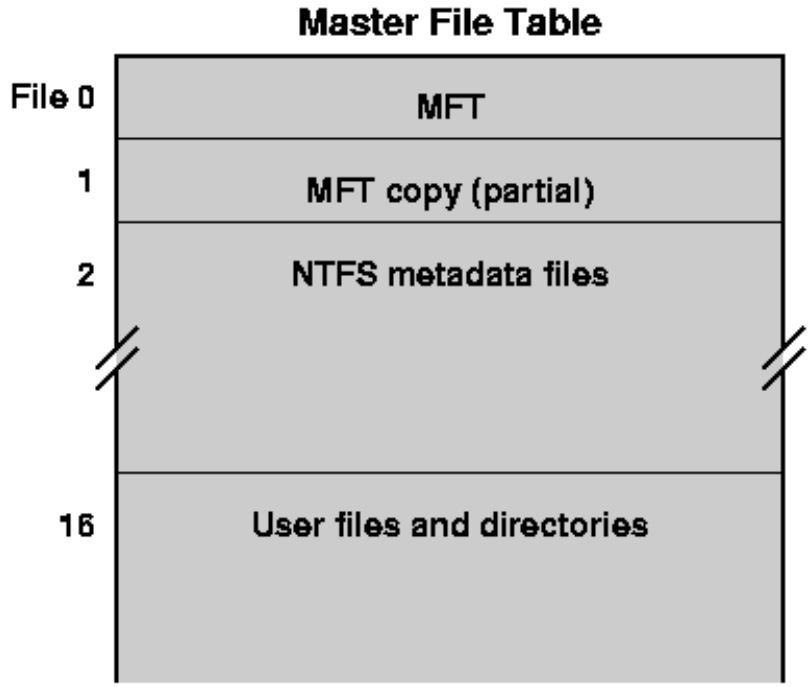
Clusters are the key element to allocation:

- Logically, the disk consists of allocation units called clusters.
- A cluster is a power-of-two multiple of the physical disk block size. The cluster size is set when the disk is formatted. A small cluster provides a finer granularity of allocation, but may require more space to describe the file and more separate operations to transfer data to or from memory.
- The free list is a bitmap, each of whose bits describe one cluster.
- Clusters on the disk are numbered starting from zero to the maximum number of clusters (minus one). These numbers are called logical cluster numbers (LCN) and are used to name blocks (clusters) on disk.

The MFT is the major, and in some ways, the only data structure on disk:

- All files, and therefore all objects stored on disk are described by the MFT.
- All files are logically stored in the MFT and, for small files are physically within the bounds of the MFT. In this sense, the MFT is the file system.
- The MFT logically can be described as a table with one row per file.

- The first rows in the table described important configuration files, including files for the MFT itself.



MFT Entries

As stated previously, each row or entry in the MFT (called a record) describes a file and logically contains the file. In the case of small files, the entry actually contains the contents of the file.

Each entry is consists of (attribute, value) pairs. While the conceptual design of NTFS is such that this set of pairs is extensible to include user-defined attributes, current versions of NTFS have a fixed set. The main attributes are:

- Standard information: This attribute includes the information that was standard in the MS-DOS world:
    - read/write permissions,
    - creation time,
    - last modification time,

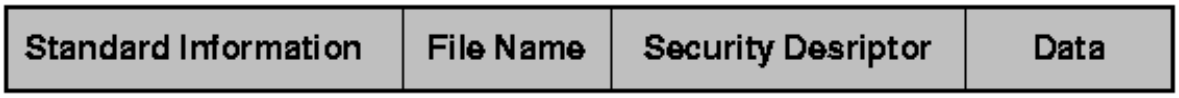
    - count of how many directories point to this file (hard link count.

- File Name: This attribute describes the file's name in the Unicode character set. Multiple file names are possible, such as when:

    - the file has multiple links, or
    - the file has an MS-DOS short name.

- Security Descriptor: This attribute lists which user owns the file and which users can access it (and how they can access it).
- Data: This attribute either contains the actual file data in the case of a small file or points to the data (or points to the objects that point to the data) in the case of larger files.

MFT Entry (Simplified)



For small files, this design is extremely efficient. By looking no further than the MFT entry, you have the complete contents of the file.

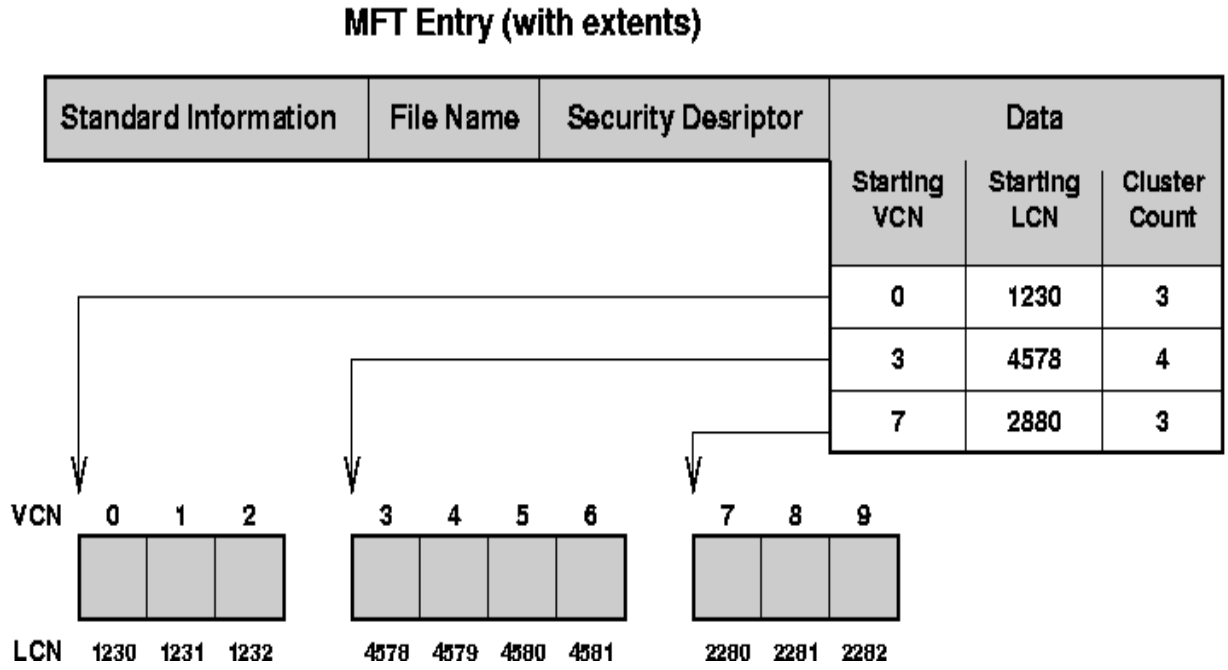
However, the Data field gets interesting when the data contained in the file is larger than an MFT entry. When dealing with large data, the Data attribute contains pointers to the data, rather than the data itself.

- The pointers to data are actually pointers to sequences of logical clusters on the disk.
- Each sequence is identified by three parts:

    - starting cluster in the file, called the virtual cluster number (VCN),
    - starting logical cluster (LCN) of the sequence on disk,
    - length, counted as the number of clusters.

- The run of clusters is called an extent, following the terminology developed by IBM in the 1960's.
- NTFS allocates new extents as necessary. When there is no more space left in the MFT entry, then another MFT entry is allocated. This design

is effectively a list of extents, rather than the Unix or DEMOS tree of extents.



Directories

As with other modern file systems, a directory in NTFS is a file whose data contains a collection of name/file mappings.

- A directory entry contains the name of the file and file reference. The file references identify the file on this volume. In other words, it is an internal name for the file.

A reference is a (file number, sequence number) pair. The file number is the offset of the file's entry in the MFT table. It is similar to the Unix inumber (Inode number).

- The list of file names in the directories is not stored in a simple list, but rather as a lexigraphically-sorted tree, called a B+ tree (this will be familiar to those with a database background). The data structure is called an index in NFTS (again, following the terminology from databases).
- The NTFS design specifies that an index can be constructed for any attribute, but currently only file name indices are supported.

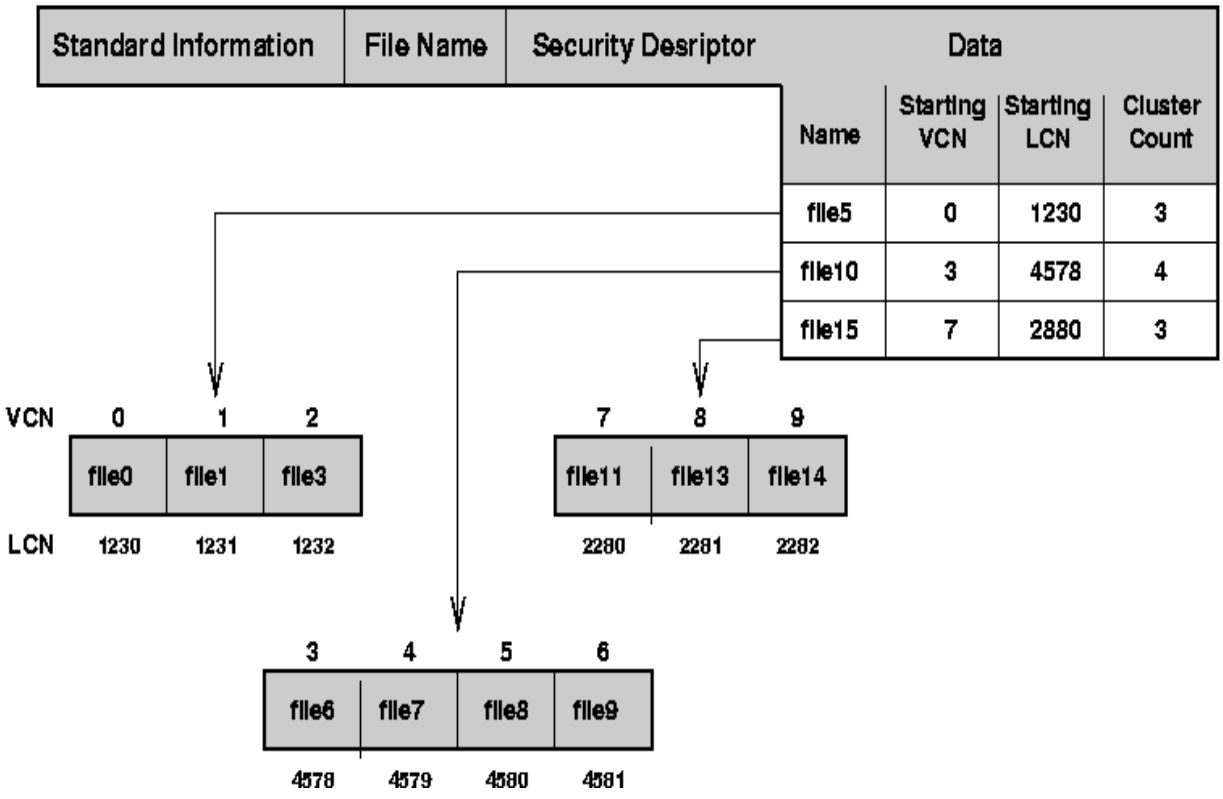
- The name for a file appears both in its directory entry and in the MFT entry for the file itself.
- As with regular files, if the directory is small enough, it can fit entirely within the MFT entry.

MFT Directory Entry (Everything Fits)

Standard Information	File Name	Security Desriptor	Index		
			file5	file10	file15

If the directory is larger, then the top part of (the B+ tree of) the directory is in the MFT entry, which points to extents that contain the rest of the name/file mappings.

MFT Directory Entry (with extents)



File System Crash Recovery

Unix File System Crash Recovery

Computers can crash at any time, and we want the file system to behave sensibly in the face of crashes. The key idea is called consistency:

- The file data and the various control structures (descriptors, bitmaps) must be in agreement.
- Since crashes can occur at any time, not all updates to the disk may be completed.
- We must insure that when the system reboots, it can return its file system to some sensible state.
- The key constraint is that any file system write operation, in progress at the time of the crash, either completely finishes or appears as if it never happened. This is called atomicity by the database folks.

Insuring consistency requires two things:

- Updates to the file system data structures must be done in the write order (and there is only one right order)!
- The proper steps must be taken at reboot time to bring the system back in to a consistent state.

There are three basic updates that happen when data is written to a file.

1. A block (or blocks) is allocated from the free list (bit map).
2. Data is written to the newly allocated block.
3. The inode is updated to include the new data.

These operations must be done in the above order. If they are not, then it is possible to have a data block included in a file that might have garbage (uninitialized data) in the block.

After rebooting, the recovery utility program on Unix, called "fsck", is going to traverse the entire directory structure of the disk to insure that all free blocks are in the free list.

Recovery after a crash follows these steps:

1. Allocate a temporary bit map, initialized to indicate that all disk blocks are free.

2. Start at the inode for the root directory.
3. Traverse the directory:

    - For each disk data block in the directory file, marks its blocks as "allocated" in the bit map.
    - For each data file in this directory, marks its data blocks as "allocated" in the bit map.
    - For each directory in this directory, perform the "Traverse the directory" steps above.

At the completion of the algorithm, you can compare the actual bit map to the temporary one to find blocks that were allocated, but never made it into a file.

Windows File System Crash Recovery

NTFS assures that the file system will remain consistent by use of a write log. This technique is similar to that used in a database system.

As in other file systems, consistency means that a write (or group of writes) to a file either complete or do not happen at all. It is not possible for a data block to be in an undefined state (e.g., allocated, but not written).

- The log is one of those standard files stored at the beginning of the MFT. It is called, cleverly enough, the log file.
- A simplified version of the steps to write data to a file look like:

    - A file update is written to the in-memory log buffer.
    - Updates to the in-memory file data and associated file system structures are made.
    - The log changes are flushed to disk.
    - The file data and structure changes are flushed to disk.

- If the system crashes during a file update, it is sufficient to go through the log an re-do each operation specified in the log.

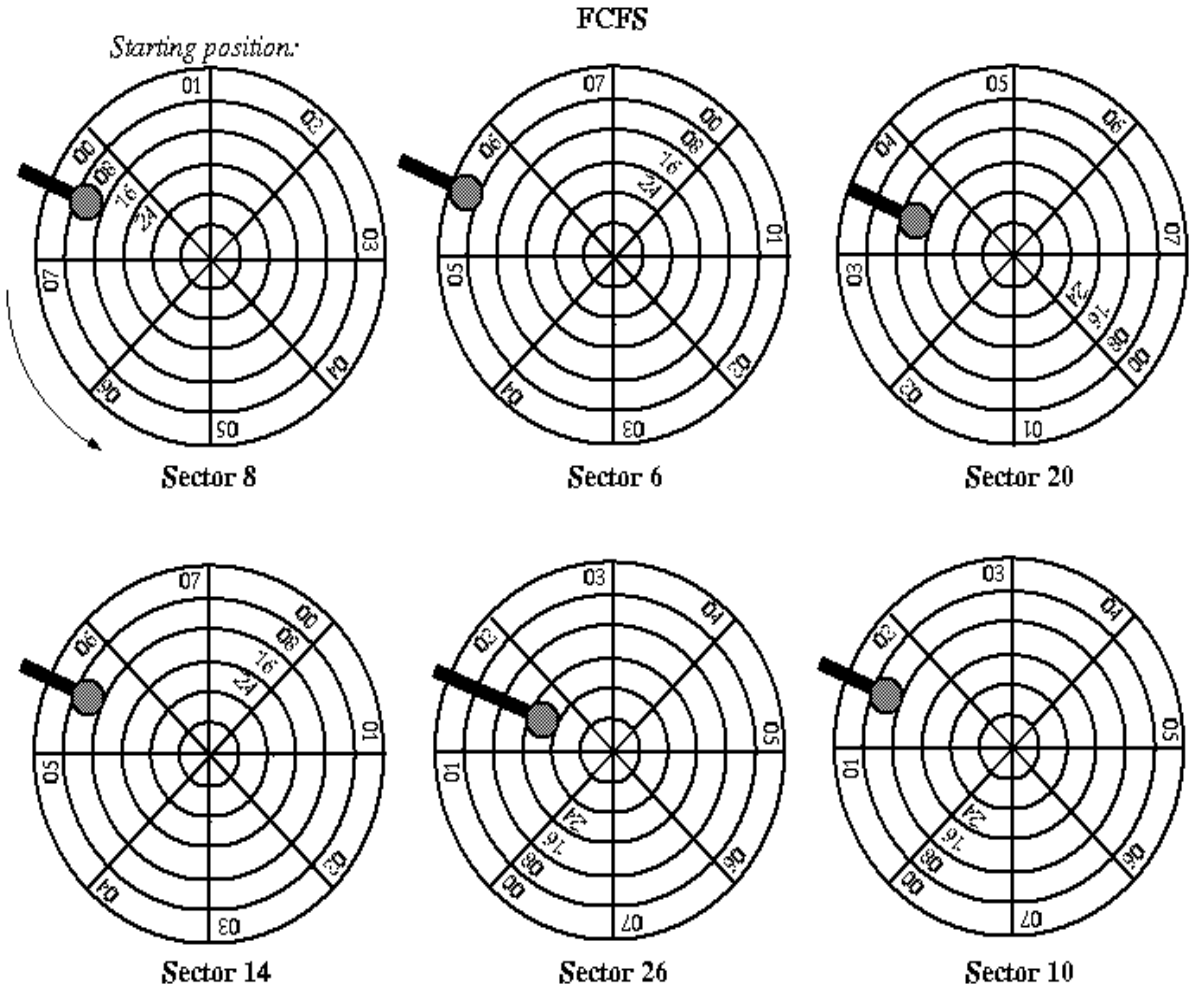
- The system occasionally creates checkpoints, so that it does not have to back to the beginning of the log for recovery. Checkpoints have two main benefits:

- Log files can be truncated, reducing the space needed for the log.
- Recovery time is faster if fewer log records need to be processed.

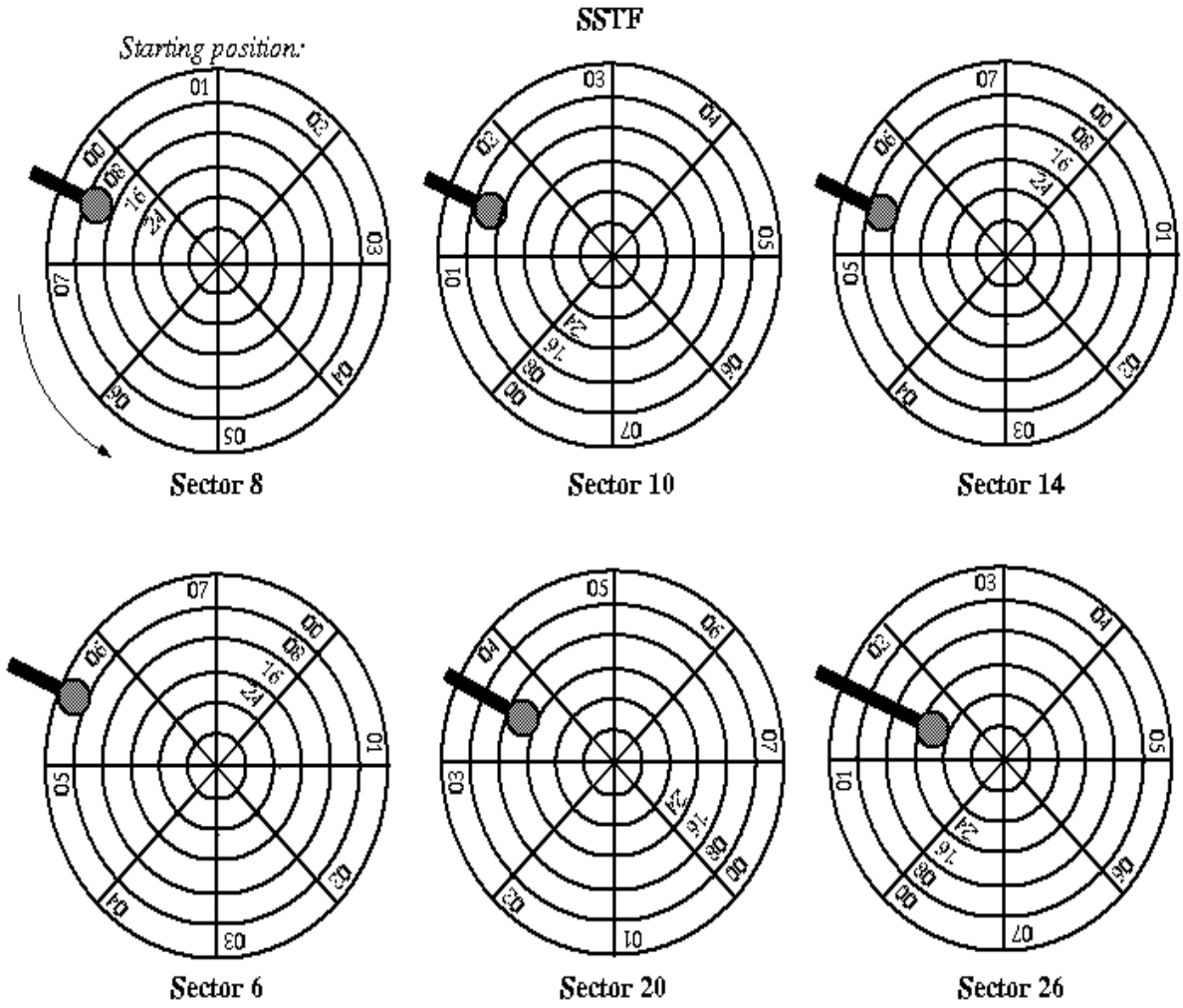
Disk Scheduling

Disk scheduling: in a system with many processes running, it can often be the case that there are several disk I/O's requested at the same time. The order in which the requests are serviced may have a strong effect on the overall performance of the disk.

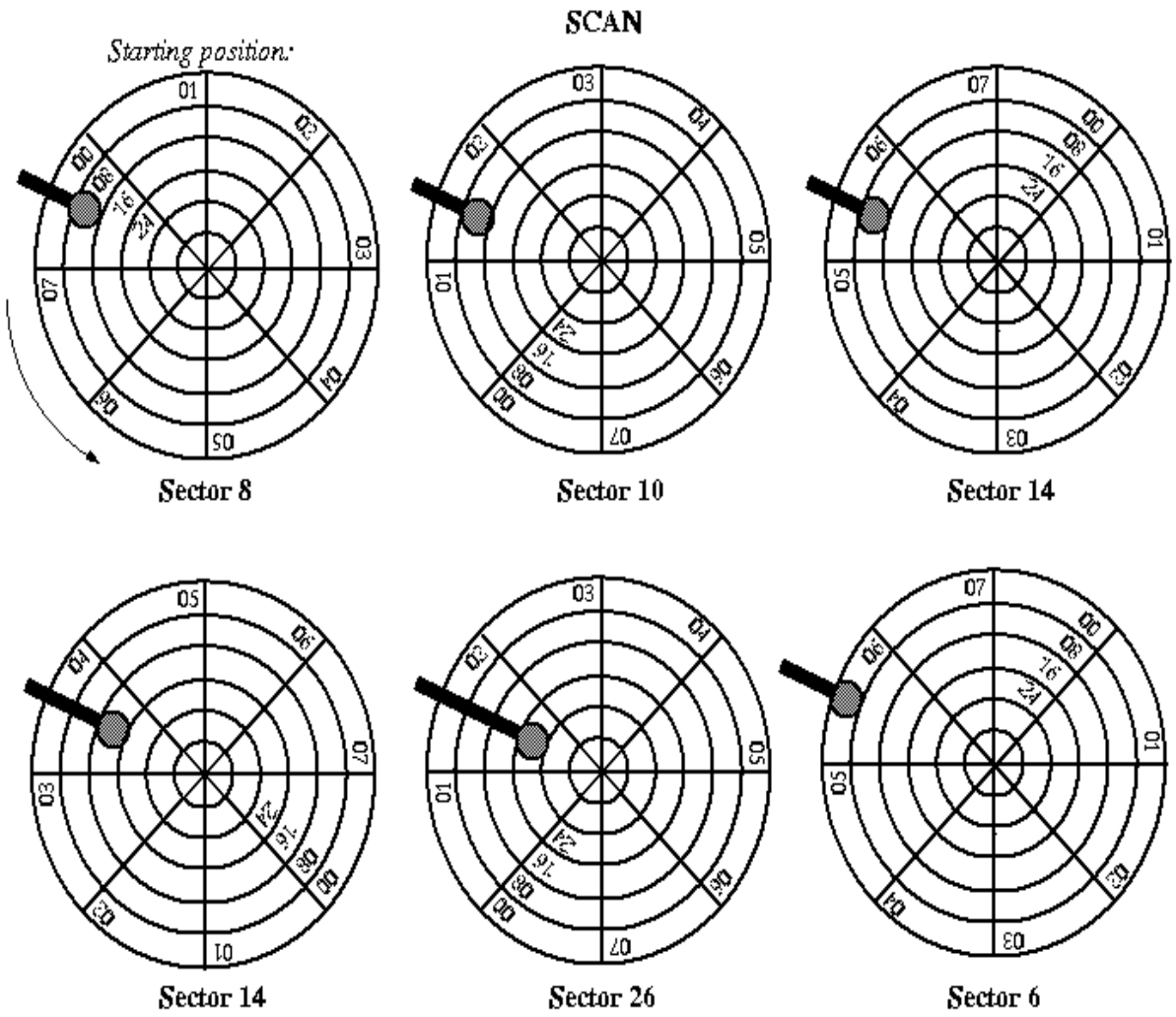
First come first served (FIFO, FCFS): may result in a lot of unnecessary disk arm motion under heavy loads.



Shortest seek time first (SSTF): handle nearest request first. This can reduce arm movement and result in greater overall disk efficiency, but some requests may have to wait a long time.



Scan: like an elevator. Move arm back and forth, handling requests as they are passed. This algorithm does not get hung up in any one place for very long. It works well under heavy load, but not as well in the middle (about 1/2 the time it will not get the shortest seek).



Minor variant: C-SCAN, which goes all the way back to front of disk when it hits the end, sort of like a raster scan in a display.

LOOK algorithm. Like scan, but reverse direction when hit the last request in the current direction. C-LOOK is the circular variant of LOOK. What most systems use in practice.

Protection and Security

Protection and Security

The purpose of a protection system is to prevent accidental or intentional misuse of a system.

- Accidents: Problems of this kind are easy to anticipate (It's possible to take action to minimize the likelihood of an accident).
- Malicious abuse: Problems of this kind are very hard to completely eliminate (In order to protect completely against malicious abuse, one must anticipate and eliminate every loophole and resist any temptation to play on probabilities).

There are three aspects to a protection mechanism:

- **User identification (authentication):** make sure we know who is doing what.
- **Authorization determination:** must figure out what the user is and is not allowed to do. Need a simple database for this.
- **Access enforcement:** must make sure there are no loopholes in the system.

Even the slightest flaw in any of these areas may ruin the whole protection mechanism.

Authentication

User identification is most often done with passwords. This is a relatively weak form of protection.

- A password is a secret piece of information used to establish the identity of a user.
- Passwords should not be stored in a readable form. One-way transformations should be used. A 1-way function is an interesting

function that is relatively easy to compute, but difficult to invert (essentially the only way to invert it is to compute all the forward transforms looking for one that matches the reverse).
- Passwords should be relatively long and obscure.

- Systems like UNIX(R) don't store the password, but the result of a 1-way function on the password. To check a user's password, the system takes the password as input, computes the 1-way function on it, and compares it with the result in the password file. If they match, the password was (with high probability) correct. Note that even knowing the algorithm and the encrypted password, it's still impossible to easily invert the function.

Although it's theoretically reasonable to leave a hashed password file in the open, it is rarely done anymore. There are a couple reasons:

- In practice, bad passwords are not uncommon enough, so rather than having to try all the passwords (or half the passwords on average), trying a large dictionary of common passwords is often enough to break into an account on the system.
- Password file can be attacked off-line, with the system under attack completely unaware that it is under attack. By forcing the attacker to actually try passwords on the system that they're invading, the system can detect an attack.

Another form of identification: badge or key.

- Does not have to be kept secret.
- Should not be able to be forged or copied.
- Can be stolen, but the owner should know if it is.

Key paradox: key must be cheap to make, hard to duplicate. This means there must be some trick (i.e. secret) that has to be protected.

Once identification is complete, the system must be sure to protect the identity since other parts of the system will rely on it.

Authorization Determination

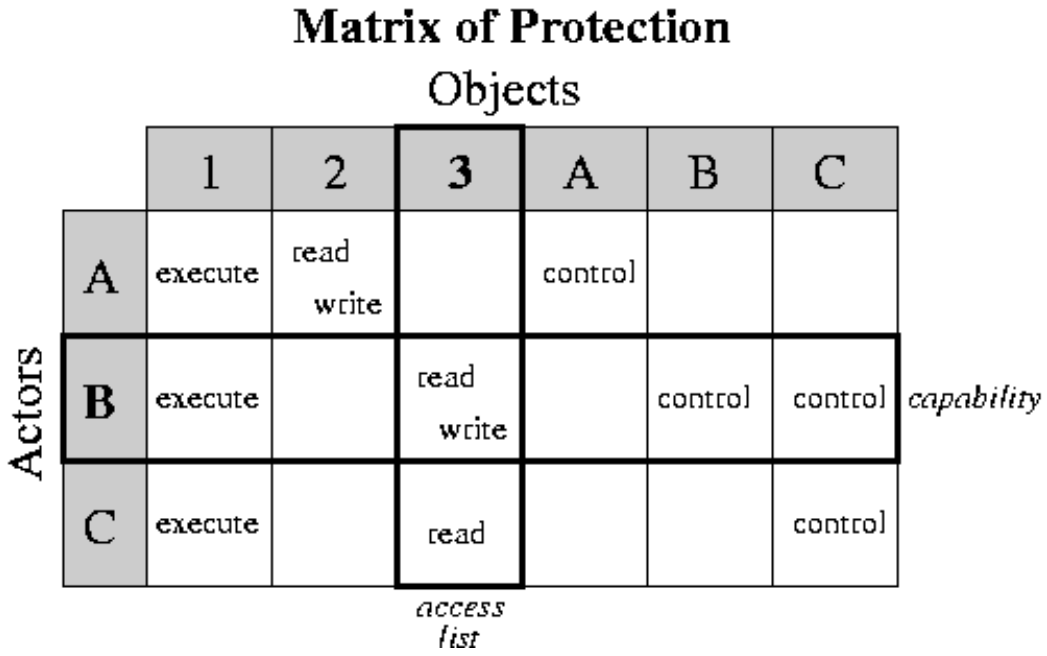
Must indicate who is allowed to do what with what. Draw the general form as an access matrix with one row per user, one column per file. Each entry indicates the privileges of that user on that object. There are two general ways of storing this information: access lists and capabilities.

Access Lists: with each file, indicate which users are allowed to perform which operations.

- In the most general form, each file has a list of pairs.
- It would be tedious to have a separate listing for every user, so they are usually grouped into classes. For example, in Unix there are three classes: self, group, anybody else (nine bits per file).
- Access lists are simple, and are used in almost all file systems.

Capabilities: with each user, indicate which files may be accessed, and in what ways.

- Store a list of pairs with each user. This is called a capability list.
- Typically, capability systems use a different naming arrangement, where the capabilities are the only names of objects. You cannot even name objects not referred to in your capability list.
- In access-list systems, the default is usually for everyone to be able to access a file. In capability-based systems, the default is for no-one to be able to access a file unless they have been given a capability. There is no way of even naming an object without a capability.
- Capabilities are usually used in systems that need to be very secure. However, capabilities can make it difficult to share information: nobody can get access to your stuff unless you explicitly give it to them.



Are the following things access-based or capability-based protection schemes?

- Protection Keys
- Page tables

Access Enforcement

Some part of the system must be responsible for enforcing access controls and protecting the authorization and identification information.

- Obviously, this portion of the system must run unprotected. Thus it should be as small and simple as possible. Example: the portion of the system that sets up memory mapping tables.
- The portion of the system that provides and enforces protection is called the security kernel. Most systems, like Unix, do not have a security kernel. As a consequence, the systems are not very secure.

- What is needed is a hierarchy of levels of protection, with each level getting the minimum privilege necessary to do its job. However, this is likely to be slow (crossing levels takes time).

File System Security

The problem addressed by the security system is how are information and resources protected from people. Issues include the contents of data files which are a privacy issue, and the use of resources, which is an accounting issue. Security must pervade the system, or the system is insecure, but the file system is a particularly good place to discuss security because its protection mechanisms are visible, and the things it protects are very concrete (for a computer system).

We're talking about some interesting stuff when we talk about security. For certain people who like puzzles, finding loopholes in security systems and understanding them to the point of breaking them is a challenge. I understand the lure of this. Remember, however, that everyone using these machines is a student like yourself who deserves the same respect that you do. Breaking into another person's files is like breaking into their home, and should not be taken lightly either by those breaking in, or those who catch them. Uninvited intrusions should be dealt with harshly (for example, it's a felony to break into a machine that stores medical records). If you really want to play around with UNIX(R) security, get yourself a linux box and play to your heart's content; don't break into someone's account here and start deleting files.

Policies and Mechanisms

Policies are real world statements about the protection that the system provides. These are all statements of (significantly different) policies:

- Users should not be able to read each other's mail
- No student should be able to see answer keys before they are made public
- All users should have access to all data.

The various systems in a computer system that control access to resources are the mechanisms that are used to implement a policy. A good security

system is one with clearly stated policy objectives that have been effectively translated into mechanisms.

The fact that data security does not stop with computer security cannot be understated. If your computer is perfectly secure, and an employee photocopies printouts of your new chip design, don't blame the computer security system.

Design Principles

Although every security system is different, some overriding principles make sense. Here is a list generated by Saltzer and Schroeder from their experience on MULTICS that remain valid today (these are fun to apply to caper movies - next time you watch Mission Impossible or Sneakers or War Games, try to spot the security flaws that let the intruders work their magic):

- Public Design Surprisingly public designs tend to be more secure than private ones. The reason is that the security community as a while reviews them and reports flaws that can be fixed. Even if you take pains to keep the source code of your system secret, you should assume that attackers have access to your code. The bad guys will share knowledge, the good guys should, too.
- Default access is no access. This holds for subsystems just like login screens. It sounds like a platitude, but is a principle worth following at all levels. People who need a certain access will let you know about it quickly.
- Test for current authority Just because the user had the right to perform an operation a millisecond ago doesn't mean they can do it now. Test the authority every time so that revocation of that authority is meaningful.
- Give each entity the least privilege required for it to do its job. This may mean creating a bunch of fine-grained privilege levels. The more privilege an entity possesses the more costly a mistake or misuse of that entity is. Printer daemons that run as root can cause logins that run as root.

- Build in security from the start. Adding security later almost never works. There are too many holes to plug, and as a practical matter security is nearly impossible to add to a fundamentally insecure system.
- In order to make such a design integration, it must be simple and capable of being applied uniformly.
- The system must be acceptable to the users. All security systems are a compromise between security and usability. The more features a system has, the more likely opportunities there are for exploitation. Furthermore, if a security feature is too onerous to the users, they will just invent ways to circumvent them. These circumventions are then available for the attackers. An unacceptable security system is automatically attacked from within.

A Sampling of Protection Mechanisms

The idea of protection domains originated with Multics and is a key one for understanding computer security. Imagine a matrix of all protection domains on one axis and all system resources (files) on another. The contents of each cell in the matrix are the operations permitted by a process (or thread) in that domain on that process.

Domain	File 1	File 2	Domain 1	Domain 2
1	RW	RWX	-	Enter
2	R	-	-	-

Notice that once domains are defined, the ability to change domains becomes another part of the domain system. Processes in given domains are allowed to enter other domains. A process's initial domain is a function of the user who starts the process and the process itself.

While the pure domain model makes protection easy to understand, it is almost never implemented. Holding the domains as a matrix doesn't scale.

Some Domains and Rings

UNIX divides processes into 2 parts, a user part and a kernel part. When running as a user the process has limited abilities, and to access hardware, it has to tap into the kernel. The kernel can access all OS and hardware, and decides what it will do on a user's behalf based on credentials stored in the PCB.

This is a simplification of the MULTICS system of protection rings. Rather than 2 levels, MULTICS had a 64 ring system where each ring was more privileged than the ones surrounding it, and checked similar credentials before using its increased powers.

Security Improvements, Encryption

Security Improvements

Solutions: nothing works perfectly, but here are some possibilities:

- Logging: record all important actions and uses of privilege in an indelible file. Can be used to catch imposters during their initial attempts and failures. E.g. record all attempts to specify an incorrect password, all super-user logins. Even better is to get humans involved at key steps (this is one of the solutions for EFT).
- Principle of minimum privilege ("need-to-know" principle): each piece of the system has access to the minimum amount of information, for the minimum possible amount of time. E.g. file system cannot touch

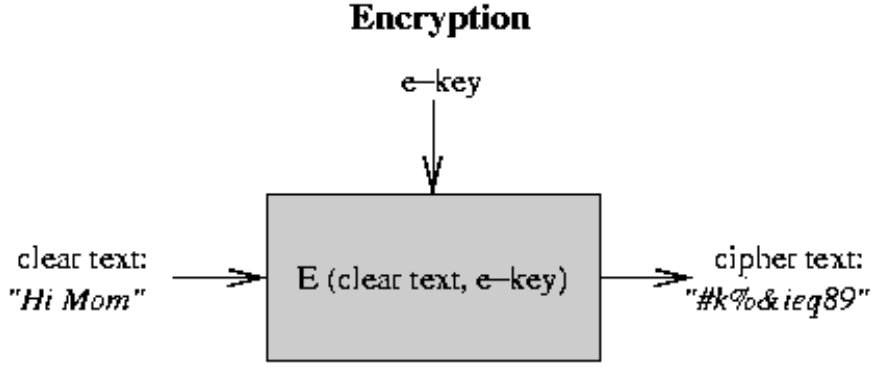
memory map, memory manager cannot touch disk allocation tables. This reduces the chances of accidental or intentional damage. Note that capabilities are an implementation of this idea. It is very hard to provide fool-proof information containment: e.g. a trojan horse could write characters to a tty, or take page faults, in Morse code, as a signal to another process.
- Correctness proofs. These are very hard to do. Even so, this only proves that the system works according to spec. It does not mean that the spec. is necessarily right, and it does not deal with Trojan Horses.

Encryption

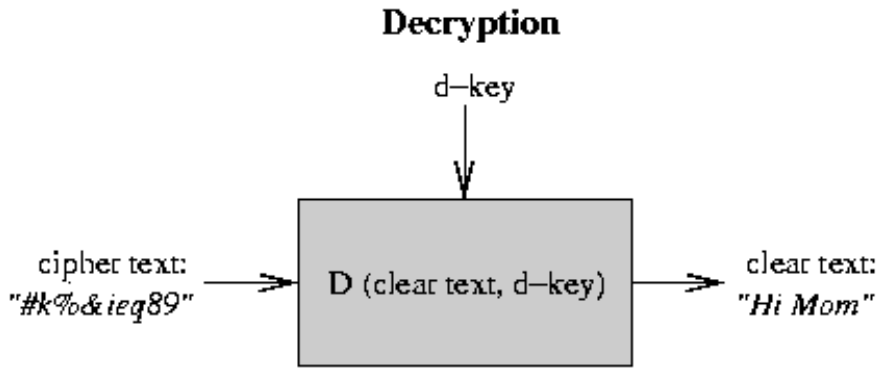
Key technology: encryption. Store and transmit information in an encoded form that does not make any sense.

The basic mechanism:

- Start with text to be protected. Initial readable text is called clear text.
- Encrypt the clear text so that it does not make any sense at all. The nonsense text is called cipher text. The encryption is controlled by a secret password or number; this is called the encryption key.



- The encrypted text can be stored in a readable file, or transmitted over unprotected channels.
- To make sense of the cipher text, it must be decrypted back into clear text. This is done with some other algorithm that uses another secret password or number, called the decryption key.



All of this only works under three conditions:

- The encryption function cannot easily be inverted (cannot get back to clear text unless you know the decryption key).
- The encryption and decryption must be done in some safe place so the clear text cannot be stolen.
- The keys must be protected. In most systems, can compute one key from the other (sometimes the encryption and decryption keys are identical), so cannot afford to let either key leak out.

Public key encryption: new mechanism for encryption where knowing the encryption key does not help you to find decryption key, or vice versa.

- User provides a single password, system uses it to generate two keys (use a one-way function, so cannot derive password from either key).
- In these systems, keys are inverses of each other: each one could just as easily encrypt with decryption key and then use encryption key to recover clear text.
- Each user keeps one key secret, publicizes the other. Cannot derive private key from public. Public keys are made available to everyone, in a phone book for example.

Safe mail:

- Use public key of destination user to encrypt mail.
- Anybody can encrypt mail for this user and be certain that only the user will be able to decipher it.

It is a nice scheme because the user only has to remember one key, and all senders can use the same key. However, how does receiver know for sure who it is getting mail from?

Digital Signatures

Positive identification: can also use public keys to certify identity:

- To certify your identity, use your private key to encrypt a text message, e.g. "I agree to pay Mary Wallace $100 per year for the duration of life."
- You can give the encrypted message to anybody, and they can certify that it came from you by seeing if it decrypts with your public key. Anything that decrypts into readable text with your public key must have come from you! This can be made legally binding as a form of electronic signature.

This is really even better than signatures: harder to forge, and can change if compromised. Note that this idea was developed by an undergraduate (Loren Kohnfelder) in 1978 as part of his undergraduate thesis (at MIT).

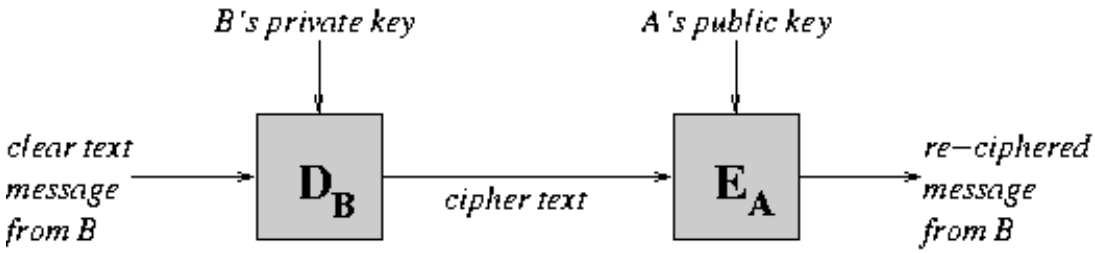
These two forms of encryption can be combined together. To identify sender in secure mail, encrypt first with your private key, then with receiver's public key. The encryption/decryption functions to send from B to A are:

$$\textbf{Encrypted text} = E(D(\text{clear text}, d\text{-key}_B), e\text{-key}_A).$$

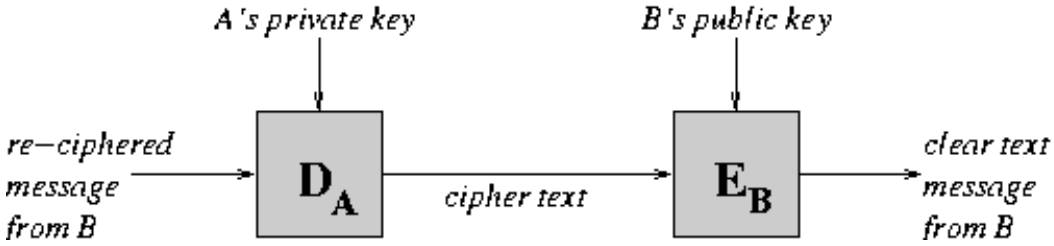
$$\textbf{Decrypted text} = E(D(\text{clear text}, d\text{-key}_A), e\text{-key}_B).$$

Digital Signatures

to send:



to receive:



Encryption appears to be a great way to thwart listeners. It does not help with Trojan Horses, though.

Old federal Data Encryption Standard (DES). Is not public-key based, but has been implemented efficiently and appears to be relatively safe.

New Advanced Encryption Standard (AES), called Ryndal.

General problem: how do we know that an encryption mechanism is safe? It is extremely hard to prove. This is a hot topic for research: theorists are trying to find provably hard problems, and use them for proving safety of encryption.

Summary of Protection: very hard, but is increasingly important as things like electronic funds transfer become more and more prevalent.

Acknowledgment

We would like to thank Prof. Barton Miller for sharing as well as for his valued advice

Sample Exams

1. [20 points, 1 each] True or False, circle T or F.

a. A binary semaphore takes on numerical values 0 and 1 only.

b. An atomic operation is a machine instruction or a sequence of instructions

that must be executed to completion without interruption.

c. Deadlock is a situation in which two or more processes (or threads) are waiting for an event that will occur in the future

d. Starvation is a situation in which a process is denied access to a resource

because of the competitive activity of other, possibly unrelated, processes.

e. While a process is blocked on a semaphore's queue, it is engaged in busy waiting.

f. Circular waiting is a necessary condition for deadlock, but not a sufficient

condition a condition for the deadlock to occur.

g. Mutual exclusion can be enforced with a general semaphore whose initial value is greater than 1

h. External fragmentation can occur in a paged virtual memory system.

i. External fragmentation can be prevented (almost completely) by frequent use of compaction, but the cost would be too high for most systems.

j. A page frame is a portion of main memory.

k. Once a virtual memory page is locked into main memory, it cannot be written to the disk.

l. Pages that are shared between two or more processes can never be swapped out to the disk.

m. The allocated portions of memory using a buddy system are all the same size.

n. Demand paging requires the programmer to take specific action to force the

operating system to load a particular virtual memory page.

o. Prepaging is one possibility for the fetch policy in a virtual memory system.

p. The resident set of a process can be changed in response to actions by other processes.

q. The working set of a process can be changed in response to actions by other processes.

r. The translation lookaside buffer is a software data structure that supports he virtual memory address translation operation.

s. In a symmetric multiprocessor, threads can always be run on any processor.

t. Thrashing will never be a problem if the system has 1 GB of real memory.

2. [20 points, 5 each] Short answers and simple diagrams.

(a) Define the resident set of a process.

(b) Define the working set of a process.

(c) What problems could occur if virtual memory pages are always allocated in groups of four?

(d) What information is used by the Least Recently Used page replacement policy, and how does this compare to the information used by the various Clock algorithms?

3. [20 points, 5 each] Short answers and simple diagrams.

(a) In terms of memory allocation, what is a reference counter? Why is it needed?

(b) Explain why, or why not, internal fragmentation can be a problem when using the best fit algorithm for memory allocation.

(c) One of the options in a mainframe OS is to limit the number of jobs (processes) currently in the system. What are some of the benefits of this capability?

(d) In what circumstances of virtual memory is the placement policy an important issue?

4. [20 points, 5 each] Short answers.

(a) What are four general characteristics of processor scheduling policies?

(b) Define Turnaround Time and Normalized Turnaround Time. Why are these useful for measuring the performance of a scheduling algorithm?

(c) What would be the effect of a large number of page faults by a process on that process's page allocation on a nonpreemptive operating system?

(d) What are four actions or decisions that a preemptive virtual memory operating system would make at the end of a time quantum (in response to a timer interrupt)?

5. [20 points]

This function is proposed for use in an operating system, with the definitions of

Process, Process_Set and other functions given elsewhere.

```
Process next_process(Process_Set available_processes) {

Process_Set A = highest_valuation(available_processes); /* priority ranking
*/

Process_Set B = earliest(A); /* actual arrival time */

Process c = random_selection(B); /* tie-breaker */

return c; /* run this process next */

}
```

(a) [5] Explain why this function could lead to processor starvation among the available processes.

(b) [5] Suppose one of the criteria used by the highest_valuation function is the process's fraction of virtual memory pages currently in main memory. Explain why this is not a good idea.

(c) [10] Define a version of the highest_valuation function (in the same style, but with some more descriptive comments) for the Shortest Process Next scheduling policy. Describe the data requirements and how this data is obtained.

Project 1: Exceptions and Simple System Calls
Operating Systems One-month project

The first project is designed to further your understanding of the relationship between the operating system and user programs. In this assignment, you will implement simple system call traps. In Nachos, an exception handler handles all system calls. You are to handle user program run time exceptions as well as system calls for IO processing. We give you some of the code you need; your job is to complete the system and enhance it.

Phase 1: Understand the Code

The first step is to read and understand the part of the system we have written for you. Our code can run a single user-level 'C' program at a time. As a test case, we've provided you with a trivial user program, 'halt'; all halt does is to turn around and ask the operating system to shut the machine down. Run the program 'nachos –rs 1023 -x ../test/halt'. As before, trace what happens as the user program gets loaded, runs, and invokes a system call.

The files for this assignment are:

progtest.cc : test routines for running user programs.

syscall.h : the system call interface: kernel procedures that user programs can invoke.

exception.cc : the handler for system calls and other user-level exceptions, such as page faults. In the code we supply, only the 'halt' system call is supported.

bitmap.* : routines for manipulating bitmaps (this might be useful for keeping track of physical page frames)

filesys.h : defines all file operations

openfile.h : (found in the filesys directory) a stub defining the Nachos file system routines. For this assignment, we have implemented the Nachos file system by making the corresponding calls to the UNIX file system directly. Because the calls are made directly, it’s necessary to debug only one thing at a time. In assignment four, we'll implement the Nachos file system for real on a simulated disk

translate.* : translation table routines. In the code we supply, we assume that every virtual address is the same as its physical address -- this restricts us to running one user program at a time. You will generalize this to allow multiple user programs to be run concurrently in a later lab.

machine.* : emulates the part of the machine that executes user programs like main memory, processor registers, etc.

mipssim.cc : emulates the integer instruction set of a MIPS R2/3000 processor.

console.* : emulates a terminal device using UNIX files. A terminal is byte-oriented and allows incoming bytes to be read and written at the same time. Bytes arrive asynchronously –as a result of user keystrokes—without being explicitly requested.

synchconsole.* : routine to synchronize lines of I/O in Nachos. Use the synchconsole class to ensure that your lines of text from your programs are not intermixed.

../test/* : C programs that will be cross-compiled to MIPS and run in Nachos

Phase 2: Design Considerations

In order to fully realize how an operating system works, it is important to understand the distinction between kernel (system space) and user space. Each process in a system has its own local information, including program counters, registers, stack pointers, and file system handles. Although the user program has access to many of the local pieces of information, the operating system controls the access. The operating system is responsible

for ensuring that any user program request to the kernel does not cause the operating system to crash. The transfer of control from the user level program to the system call occurs through the use of a “system call” or “software interrupt/trap”. Before invoking the transfer from the user to the kernel, any information that needs to be transferred from the user program to the system call must be loaded into the registers of the CPU. For pass by value items, this process merely involves placing the value into the register. For pass by reference items, the value placed into the register is known as a “user space pointer”. Since the user space pointer has no meaning to the kernel, we will have to translate the contents of the user space into the kernel such that we can manipulate the information. When returning information from a system call to the user space, information must be placed in the CPU registers to indicate either the success of the system call or the appropriate return value.

Nachos gives you a simulated CPU that models a real CPU. In fact, the simulated CPU is the same as the real CPU (a MIPS chip), but we cannot just run user programs as regular UNIX processes, because we want complete control over how many instructions are executed at a time, how the address spaces work, and how interrupts and exceptions (including system calls) are handled.

Nachos provided simulator can run normal programs compiled from C -- see the Makefile in the ‘test’ subdirectory for an example. The compiled programs must be linked with some special flags, then converted into Nachos format, using the program “coff2noff”

Phase 3: [80%] Exceptions and IO System Calls + [10%] Correct design and implementation of encrypted file types

Implement exception handling and handle the basic system calls for file IO. (All system calls are listed in syscall.h) We have provided you an assembly-language routine, ‘syscall’, to provide a way of invoking a system call from a C routine (UNIX has something similar -- try ‘man syscall’). You will need to do the following steps. NOTE : You should - NOT alter the code within the machine directory, only the code within the userprog directory.

- Alter exception.cc to handle all of system exceptions as listed in machine/machine.h Most of the exceptions listed in this file are comprised of run time errors, from which the user program will be unable to recover. The only special cases are no exception, which will return control to the operating system and syscall exception, which will handle our user system calls. For all other exceptions, the operating system should print an error message and Halt the simulation.
- Create a control structure that can handle the various Nachos system calls. Test your control structure by re-implementing the void Halt() system call. Make sure that this call operates in the same manner as we discussed during the Nachos walkthrough; it should cause the Nachos simulation to terminate immediately. Test the call's accuracy with the test user program.
- All system calls beyond Halt() will require that Nachos increment the program counter before the system call returns. If this is not properly done, Nachos will execute the system call forever. Since the MIPS emulator handles look ahead program counters, as well as normal ones, you will have to emulate the program counter increment code as found in the machine directory. You will have to copy the code into your syscall exception handler and insert it at the proper place. For now, you will have to use the Halt() system call at the end of each of your user programs.
- Implement the int CreateFile(char *name) system call. The createfile system call will use the Nachos Filesystem Object Instance to create a zero length file. Remember, the filename exists in user space. This means the buffer that the user space pointer points to must be translated from user memory space to system memory space. The createfile system call returns 0 for successful completion, -1 for an error.
- Implement the OpenFileID Open(char *name, int type) and int Close(OpenFileID id) system calls. The user program can open two types of "files", files that can be read only and files that can be read and write. Each process will allocate a fixed size file descriptor table. For now, set this size to be 10 file descriptors. The first two file descriptors, 0 and 1, will be reserved for console input and console output respectively. The open file system call will be responsible for

translating the user space buffers when necessary and allocating the appropriate kernel constructs. You will use the filesystem objects provided to you in the filesystem directory. (NOTE: We are using the FILESYSTEM_STUB code) The calls will use the Nachos Filesystem Object Instance to open and close files. The Open system call returns the file descriptor id (OpenFileID == an integer number), or –1 if the call fails. Open can fail for several reasons, such as trying to open a file or mailbox that does not exist or if there is not enough room in the file descriptor table. The type parameter will be set to 0 for a standard file and 1 for a read only file, 2 for encrypted files. Encrypted files will have modified read and writes that encrypt the content. If the type parameter is set to any other value, the system call should fail. The close system call will take a file descriptor as the parameter. The system call will return –1 on failure and 0 on success.

- For crypto files any simple encryption /decryption scheme can be used. E.g. a separate character mapping table can be maintained that maps one character to a different character. Write method will use this mapping table for encryption and read method will use the inverse of the mapping table for decryption.
- Implement the int Read(char *buffer, int charcount, OpenFileID id) and int Write(char *buffer, int charcount, OpenFileID id) system calls. These system calls respectively read and write to a file descriptor ID. Remember, you must translate the character buffers appropriately and you must differentiate between console IO (OpenFileID 0 and 1) and File (any other valid OpenFileID). The read and write interaction will work as follows:

For console read and write, you will use the SynchConsole class, instantiated through the gSynchConsole global variable. You will use the default SynchConsole behaviors for read and write, however you will be responsible for returning the correct types of values to the user. Read and write to Console will return the actual number of characters read or written, not the requested number of characters. In the case of read or write failure to console, the return value should be –1. End of file from the console is returned when the user types in Control-A. Read and write for console will use ASCII data for input and output. (Remember, ASCII data is NULL (\0) terminated)

For file read and write, you will use the supplied classes in file system. You will use the default filesystem behaviors, however, you will return the same type of return values as for synchconsole. Both read and write will return the actual number of characters read and written. Both system calls will return –1 for failure. Read and write for files will use binary data for input and output.

- Implement the int Seek(int pos, OpenFileID id) system call. Seek will move the file cursor to a specified location. The parameter pos will be the absolute character position within a file. If pos is a –1, the position will be moved to the end of file. The system call will return the actual file position upon success, -1 if the call fails. Seeks on console IO will fail.
- Implement a createfile user program to test the createfile system call. You are not going to pass command line arguments to the call, so you will have to either use a fixed filename, or prompt the user for one when you have console IO working.
- Implement a help user program. All help does is it prints a list to standard output of all the user programs you are going to create or have created in the test directory. Help should list each program and a brief 1 line description of each program. (NOTE: This is nothing fancy, just calls using the Write() system call to standard output.)
- Implement an echo user program. For each line of input from the console, the line gets echoed back to the console.
- Implement a cat user program. Ask for a filename, and display the contents of the file to the console.
- Implement a copy user program. Ask for a source and destination filename and copy the file.
- Implement a reverse user program. This program asks for a source and destination file, takes input from the source file, reverses the source file and writes it to the destination.
- Implement encrypt that will take a simple character input file and create a new encrypted file.
- Implement decrypt that take an encrypted file and create a new decrypted file.
- Implement any other tests you feel are necessary to ensure the correctness of your solution. BIG HINT: Implement tests to cover

some of the things we changed that are not tested by parts h through m.

NOTE: A large portion of your grade will depend not only on the correctness of your implementation but how accurately your code conforms to the system call specifications presented in this document. As we are building a robust operating system, the user should not be able to perform any system call that will crash Nachos. The worse case scenario is that the user program might generate a Nachos runtime exception, which you will be handling within part a.

Phase 4: [10%] Documentation

This includes internal documentation (comments) and a BRIEF, BUT COMPLETE external document (read as: paper) describing what you did to the code and why you made your choices. Everybody needs to demo their project (whether completed or not) to Teaching Assistants (TAs) to get a grade for the project. Demonstration times will be posted after the due date. Submit the documentation online on the due date and take a hardcopy with when you demo the project to the TA for grading. DO NOT PARAPHRASE THIS LAB DESCRIPTION AND DO NOT TURN IN A PRINTOUT OF YOUR CODE AS THE EXTERNAL DOCUMENTATION

Deliverables and grading

When you complete your project, remove all executables and object files. If you want me to read a message with your code, create a README.NOW file and place it in the nachos code directory. Tar and compress the code, and submit the file using the online submission system. It is important that you follow the design guidelines presented for the system calls. I will be running my own shells and test programs against your code to determine how accurately you designed your lab, and how robust your answers are. Grading for the implementation portion will depend on how robust and how accurate your solution is. Remember, the user should not be able to do anything to corrupt the system, and system calls should trap as many error conditions as possible.

Project 2: Multi-programming and Inter-process communication
Operating Systems One-month project

In the second project you will design and implement appropriate support for multiprogramming. You will extend the system calls to handle process management and inter-process communication primitives. You will add this to the coded first project. Make sure you correct all the deficiencies in your first project before starting the second project. This solution for project1 will be covered as part of next week's recitation.

Nachos is currently a uniprogramming environment. We will have to alter Nachos so that each process is maintained in its own system thread. We will have to take care of memory allocation and de-allocation. We will also consider all the data and synchronization dependencies between threads. You will first design the solution before coding. Here are the details:

1. Alter your general exceptions (non-system call exceptions) to finish the thread instead of halting the system. This will be important, as a run time exception should not cause the operating system to shut down. You will most likely have to revisit this code several times before your project is complete. There are several synchronization issues you will have to handle during thread exit.
2. Implement multiprogramming. The code we have given you is restricted to running one user program at a time. You will need to make some changes to addrspace.h, and addrspace.cc in order to convert the system from uniprogramming to multiprogramming. You will need to:

    - Come up with a way of allocating physical memory frames so that multiple programs can be loaded into memory at once.
    - Provide a way of copying data to/from the kernel from/to the user's virtual address space.
    - Properly handling freeing address space when a user program finishes.
    - It is very important to alter the user program loader algorithm such that it handles frames of information. Currently, memory space allocation assumes that a process is loaded into a contiguous section of memory. Once multiprogramming is active,

memory will no longer appear contiguous in nature. If you do not correct the routine, it is most likely that loading another user program will corrupt the operating system.

3. Implement the **SpaceID Exec(char *name)** system call. Exec starts a new user program running within a new system thread. You will need to examine the "StartProcess" function in progtest.cc in order to figure out how to set up user space inside a system thread. Exec should return –1 on failure; else it should return the "Process SpaceID" of the user level program it just created. (Note: SpaceIDs can be kept track of in a similar manner to OpenFileIDs of your project 1, except that you will want to keep track of them outside the thread.)
4. Implement the int Join(SpaceID id) and void Exit(int exitCode) system calls. Join will wait and block on a "Process SpaceID" as noted in its parameter. Exit returns an exit code to whoever is doing a join. The exit code is 0 if a program successfully completes another value if there is an error. The exit code parameter is set via the exitcode parameter. Join returns the exit code for the process it is blocking on, -1 if the join fails. A user program can only join to processes that are directly created by the Exec system call. You can not join to other processes or to yourself. You will have to use semaphores inside your system calls to coordinate Joining and Exiting user processes. You will observe that this can be modeled as sleeping barber IPC (inter process communication).
5. Implement the int CreateSemaphore(char *name, int semval) system call. From the execv system call that you implemented in Project1 you would have realized that we will have to update start.s and syscall.h to add the new system call signatures. You will create a container at the system level that can hold upto 10 named semaphores. The CreateSemaphore system call will return 0 on success and –1 on failure. The CreateSemaphore system call will fail if there are not enough free spots in the container, the name is null, or the initial semaphore value is less than 0. (You should reuse the Semaphore class which is implemented in the source code)
6. Implement int wait(char *name) and int signal(char *name) system calls. Make sure you follow the wait and signal as the mnemonics for these two and not down and up or P and V. The name parameter is the

name of the semaphore. Both system calls return 0 on success and –1 on failure. Failure can occur if the user gives an illegal semaphore name (one that has not been created).

7. Implement a simple shell program to test your new system calls implemented as above. The shell should take a command at a time, and run the appropriate user program. The shell should “Join” on each program “Exec”ed, waiting for the program to exit. On return from the Join, print the exit code if it is anything other than 0 (normal execution). Also, design the shell such that it can run program in the background. Any command starting with character (&) should run in the background. (Ex: &create will run the test program create program in the background.)
8. Solve the Dining Philosopher problem discussed in your text book. Use Semaphore you designed for realizing mutual exclusion and synchronization among the philosophers.
9. [Tanenbaum] A student majoring in anthropology and minoring in computer science has embarked on a research project to see if African baboons can be taught about deadlocks. He locates a deep canyon and fastens a rope across it, so the baboons can cross hand-over-hand. Several baboons can cross at the same time, provided that they are all going in the same direction. If eastward moving and westward moving baboons ever get onto the rope at the same time, a deadlock will result (the baboons will get stuck in the middle) because it is impossible for one baboon to climb over another one while suspended over the canyon. If a baboon wants to cross the canyon, it must check to see that no other baboon is currently crossing in the opposite direction. Write a program using semaphores that avoids deadlock. Do not worry about a series of eastward moving baboons holding up the westward moving baboons indefinitely.
10. Documentation (10%) Includes internal documentation, and external documentation as described in Project1. Create a READE file and place in the code directory. Tar your code and submit it as one file. Follow the directions given in Project1. We plan to run automatic plagiarism detection software to detect any copied projects. The consequences are quite unpleasant for academic dishonesty. So work on your own and not in groups. This is not a group project. No late projects will be accepted.

Project 3: Chat Application Using Nachos Networking Module
Operating Systems One-month project

Objectives:

- Learn to work with Nachos networking for communication among processes.
- Define and implement a “chat” protocol for interaction between a chat server and a chat client.

Problem Statement:

Chat rooms have become a popular way to support a forum for n-way conversation or discussion among a set of people with interest in a common topic. Chat applications range from simple, text-based ones to entire virtual worlds with exotic graphics. In this project you are required to implement a simple text-based chat client/server application.

Problem Description:

Email, newsgroup and messaging applications provide means for communication among people but these are one-way mechanisms and they do not provide an easy way to carry on a real-time conversation or discussion with people involved. Chat room extends the one-way messaging concept to accommodate multi-way communication among a set of people.

Nachos networking infrastructure

Nachos networking packages implements a very simple Unix domain (not internet domain), datagram socket. The files of importance to this project and their purpose are described below.

threads directory:

lockcond.h, lockcond.cc: for defining and implementing lock and condition synchronization primitives; needed for network applications to work.

machine directory:

network.h, network.cc: Data structures to emulate a physical network connection. The network provides the abstraction of ordered, unreliable, fixed-size packet delivery to other machines on the network.You may note that the interface to the network is similar to the console device -- both are full duplex channels.

sysdep.h, sysdep.cc : Interprocess communication operations, for simulating the network; Unix sockets creation, binding, closing etc. are called here to provide nachos socket functionality.

network directory:

post.h, post.cc: postoffice and mailbox, mail message definition and implementation.

nettest.cc: application to test communication between host id 0 and host id 1 (these are hardcoded!)

Chat Architecture:

A chat application consists of a Chat Client (CC) one per person, a Chat Server (CS) and a two-way communication pipeline between the client and the server to send and receive conversational, control and status messages.

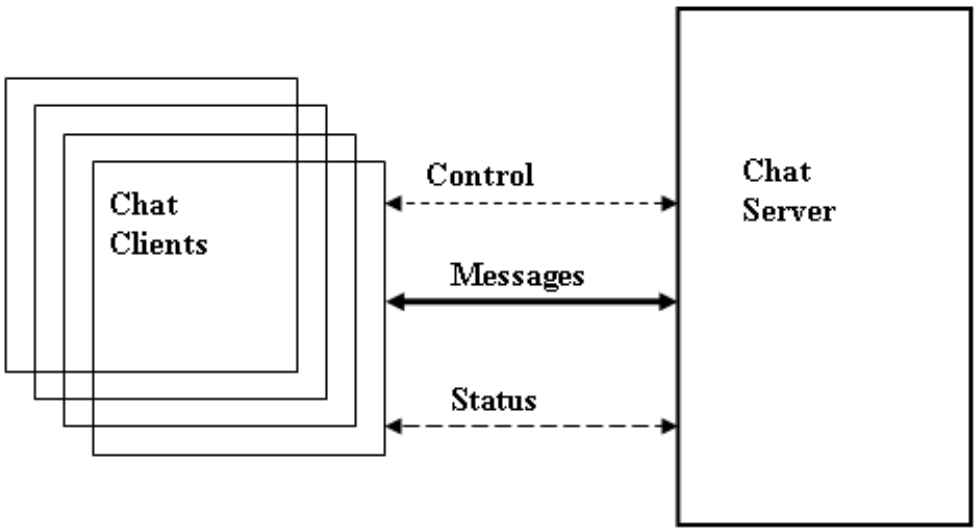


Figure 1: Chat Application Overview

- Chat Client:

Typical features of a CC include: (i) select chat server (server id), (ii) select a nickname for interaction and (iii) ability to set and change user preferences such as number of messages displayed, change nickname, etc. You may design your own chat user interface.

Implementation notes:

The server id is specified at command line (nachos –rs 1234 –m selfid –o serverid) while executing the nachos process representing the client.

Once the client process is running it could interactively ask the user for the nickname of the chat user and update this information locally and if needed by your design on the server.

- Chat Server:

A chat server supports the set of clients for a room, by maintaining client handle (clientid in nachos –rs –m clientId –o serverid), and client name. Server also has a message interpreter that parses the message received from a client and delegates the command to the appropriate module. Sample architecture for the chat application is shown in Figure 2. You may change the architecture to suit your design.

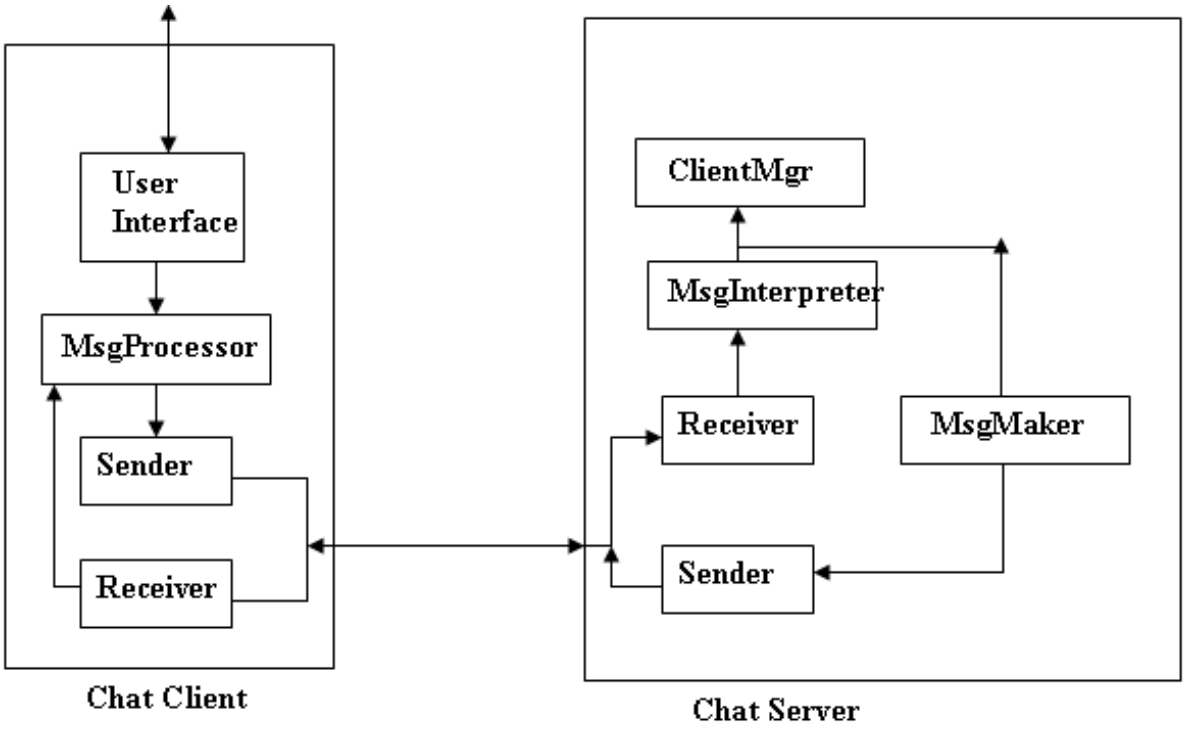


Figure 2: Chat Application Architecture

The Chat client receives the user messages and user configuration, sets up commands and passes them to the server. Sometimes it is also possible to process some of the commands (Ex: Number of messages displayed) locally. MsgProcessor in Figure 2 Chat Client is responsible for interpreting messages from the user. Sender and receiver are for communicating with the server. Messages are constructed as described in the protocol below.

A chat server receives the commands and messages from the chat clients and processes them. MsgInterpreter is for unpacking, parsing and delegating the commands to the appropriate units on the server side. MsgMaker constructs the messages to be sent back according to the protocol described below.

We will use nachos sockets for communication between the server and the client.

Chat Protocol:

Protocols such as TCP and HTTP provide rules for communication. They specify details of message formats; they describe how an application

responds when a message arrives, and how to handle abnormal and error conditions. We describe the protocol we will use for the chat service in Section 8.

Implementation Details

Phase 1: Study all the codes associated with the nachos networking.

Phase 2: Perquisites: Lock and Condition: (15%) Implement the lock and condition the skeleton for which are in threads directory. Test it. Run the nettest application in network directory by opening up two xterms/terminals and running two nachos processes (network id 0, 1) communicating with each other.

Implement lock and condition.

Go into network directory, gmake

Run nachos on two xterm/terminals using these commands:

nachos –rs 1234 –m 0 –o 1

nachos –rs 1234 –m 1 –o 0

You will observe the two processes sending messages and acknowledging.

Phase 3: Ring Network: (5%) Update the nettest.cc so that a set of nachos process with network ids (0, 1,2,3..) can communicate. To test this form a ring of at least three nachos processes representing three network nodes, node 0 sends message to node 1, node 1 receives and transmits the same message to node 2 and node 2 receives and transmits the message back to node 0 thus successfully completing a trip around the ring.

Phase 4: (45%) Chat Server: Implement the chat server that behaves according to the protocol described in Section 8. Test it with dummy data/hard coded data.

Phase 5: (25%) Chat Client and Integration with Server: Implement the chat client and a simple text interface and integrate it with the server.

Phase 6: (10%) Documentation:

This includes internal documentation (comments) and a BRIEF, BUT COMPLETE external document (read as: paper) describing what you did to the code and why you made your choices. DO NOT PARAPHRASE THIS LAB DESCRIPTION AND DO NOT TURN IN A PRINTOUT OF YOUR CODE AS THE EXTERNAL DOCUMENTATION.

Deliverables and grading

When you complete your project, remove all executables and object files. If you want me to read a message with your code, create a README.NOW file and place it in the nachos code directory. Tar and compress the code, and submit the file using the online submission system. It is important that you follow the design guidelines presented for the system calls. I will be running my own shells and test programs against your code to determine how accurately you designed your lab, and how robust your answers are. Grading for the implementation portion will depend on how robust and how accurate your solution is. Remember, the user should not be able to do anything to corrupt the system, and system calls should trap as many error conditions as possible.

8. A Simple Chat Protocol

SERVERINFO

```
cmd_serverinfo {
        byte    type    =       0;
        byte    cmd     =       0;
        };
status_serverinfo {
        byte    type    =       1;
        byte    cmd     =       0;
```

```
        byte    status  =       N;      // 0 = OK,
1 = FAIL
        byte    length  =       M;      // Size of
message
        byte[]  message;
                };

        OUTPUT: [SERVER] – Copyright 2004 Your
Name Here, CSE4/521
        OUTPUT: [SERVER] – ERROR : SERVERINFO
Command Failed
```

LOGIN <nickname>

```
cmd_login {
        byte    type    =       0;
        byte    cmd     =       1;
        byte    length  =       N;
        byte[]  nickname;
        };
status_login {
        byte    type    =       1;
        byte    cmd     =       1;
        byte    status  =       N;      // 0 = OK,
1 =FAIL

// 2 = Nickname in use

// 3 = Already logged in

        };
OUTPUT: [SERVER] – User <nickname> logged in.
OUTPUT: [SERVER] – ERROR : LOGIN error.
OUTPUT: [SERVER] – ERROR : LOGIN nickname
<nickname> in use.
```

```
OUTPUT: [SERVER] – ERROR : LOGIN user already
logged in.
```

LOGOUT

```
cmd_logout {
              byte      type    =       0;
              byte      cmd     =       2;
        };

status_logout {
        byte    type    =       1;
        byte    cmd     =       2;
        byte    status  =       N;      // 0 = OK,
1 FAIL
                                        // 2 = Not
logged in
        };

        OUTPUT: [SERVER] – User <nickname> logged
out.
        OUTPUT: [SERVER] – ERROR : LOGOUT error.
        OUTPUT: [SERVER] – ERROR : User not logged
in.
```

WHOSINROOM

```
 cmd_whoroom {
        byte    type    =       0;
        byte    cmd     =       3;
        };


status_whoroom {
        byte    type    =       1;
        byte    cmd     =       3;
```

```
        byte    status  =       N;      // 0 = OK,
1 FAIL
        };

data_whoroom {
        byte    type    =       2;
        byte    cmd     =       3;
        byte    entry   =       count;   //
nickname count
        byte    pos     =       X;      // 0 =
Middle, 1 = First, 2 = Last
        byte    length  =       L;  // message
length
        byte[]  msg;            // Single message
Entry
        };
        OUTPUT: [SERVER] – ERROR : WHOSINROOM
error.
        OUTPUT: [SERVER] – User : <X> NickName:
<nickname>
(As many data N data message representing N
nicknames will be sent)
```

SEND <message>

```
 cmd_send {
        byte    type    =       0;
        byte    cmd     =       4;
        byte    length  =       N;
        byte[]  message;
        };

status_send {
        byte    type    =       1;
        byte    cmd     =       4;
        byte    status  =       N;      // 0 = OK,
```

```
1 FAIL
                                        // 2 = Not
in room
};

bcast_send {
        byte    type    =       3;
        byte    cmd     =       4;
        byte    length  =       N;
        byte[]  msg;            // Single message
Entry
        };
OUTPUT: [SERVER] – ERROR : SEND error.
OUTPUT: [SERVER] – ERROR : SEND not in room.
OUTPUT: [<nickname>] – <msg>
Send message is sent from a client to the server,
which then broadcasts to all the clients.
```

WHISPER <nickname> <message>

```
 cmd_whisper {
        byte    type    =       0;
        byte    cmd     =       5;
        byte    nlength =       M;
        byte[]  nickname;
        byte    length  =       N;
        byte[]  message;
        };

status_whisper {
        byte    type    =       1;
        byte    cmd     =       5;
        byte    status  =       N;      // 0 = OK,
1 FAIL
                                        // 2 = Not
in room
```

```
                                                    // 3 =
nickname not found
};

bcast_whisper {
        byte    type    =       3;
        byte    cmd     =       5;
        byte    length  =       N;
        byte[]  msg;            // Single message
Entry
        };
OUTPUT: [SERVER] – ERROR : WHISPER error.
OUTPUT: [SERVER] – ERROR : WHISPER not in room.
OUTPUT: [*<nickname>*] – <msg>
```

QUIT

This command on the server side shuts the server down and cleans up all the space. (On the client side Logout command itself will terminate the server program.)

Important Note: You are required to follow strictly the protocol given. This will allow us to test your sever with our client during the demo for the project. Please do not implement any more commands than specified above. You may not have time to do that. I would rather prefer you spend your time to implement the commands given in section 8.

Project 4: Emand Paging
Operating Systems One-month project

The third Nachos project is to build a demand paging system. You will create the mechanisms and policies for a demand paging system with a simple page table system. You will also implement a simple page replacement algorithm.

Phase 1: Understand the Code

You will modify the set of files that you have after completion of **projects 1.** It is important that you understand how the demand paging and address translation system in Nachos works. To this end, we suggest that you examine the following files:

machine.* : We will not use TLB. We will compile in the vm directory to make use of the

dependency setting stated in the Makefile there. Remove USE_TLB flag from the lines stating the various components used.

translate.* : Examine the ReadMem, WriteMem, and Translate methods of machine. It is important to understand how a PageFaultException is generated for a page miss. Pay particular attention to where the invalid virtual address is stored when the exception occurs, since you will need to access it later.

addrspace.* : The role of address space will change, but it is now important to understand what ALL of the methods do, not just the constructor or deconstructor.

exception.cc : The majority of your demand paging code will be called from the PageFaultException handler.

bitmap.* : routines for manipulating bitmaps

filesys.h

openfile.h : Pay careful attention to the readAt and writeAt methods!

../test/* : C programs that will be cross-compiled to MIPS and run in Nachos

Phase 2 : Design Considerations

When the machine object is instantiated, the pagetable will be initialized with all entries invalid. This will generate a page fault when the first memory address is accessed. The majority of your design will deal with what happens when a page fault occurs.

In order to implement full demand paging, you will also be required to create and modify several other data structures. Such data structures include:

- Modification of the TranslationEntry data structure to support a reference word.
- A method for determining the source and destination expression for the page to be loaded, from a file to the main memory.

You will also have to account for all phases of the demand paging algorithm. These phases include:

- Page Table Hit

- Page Table Miss, free entry in the main memory
- Page Table Miss, page replacement in the main memory, replacement page clean, and
- Page Table Miss, page replacement in the main memory, replacement entry dirty.

You should also make sure that you separate the mechanisms and policies. Initially you will use FIFO policy for replacement. After testing pagefault handler with FIFO, replace it with LRU page replacement algorithm.

Phase 3: [90%] Lab 3 Assignment

NOTE: It is very important to compile this project in the VM directory! Your files will be in userprog.

- **Understand Page Tables –** This is very important, but will not count towards your grade.
- **Change Constants in machine.h –** Please change the constants in machine.h to the following:NumPhysPages32
- **Change class TranslationEntry in translate.h –** Add ‘int refword’ as an instance variable to this class (under the dirty bit)
- **Moving pages in and out of main memory:** A critical step in the demand paging in rolling pages needed from an executable file into the main memory and rolling out dirty pages back to the file. It is suggested that you write a helper class that will exclusively focus on these operations. For more information on the code look at address space constructor.
- **Data structure for the Page Table –** Base the design of the page table on the TranslationEntry class. You may need to add an extra entry for page replacement policy.
- **Implement FIFO Page Table Demand Paging –** You must now process all PageFaultExpections, and implement the core demand paging algorithm. After complete testing use LRU method.
- Design three user programs that can be run inside the shell to demonstrate the robustness of your overall project solution.

Phase 4: [10%] Documentation

Answer the following questions in brief:

1. Describe the steps that occur during a Page Fault Exception when: (6)
a. There are free pages in the main memory

b. There are NO free pages in the main memory & dirty bit for the replacement page is NOT set.

c. There are NO free pages in the main memory & dirty bit for the replacement page is SET.

2. Give an example of LRU & FIFO policies.(2)

3. Where exactly have you updated the reference count for the 2 policies & why specifically in these places? (Just give the function names & the reason) (2)

Project 5: Multi-programming, Inter-process Communication & Scheduling
Operation Systems One-month project

Nachos is currently an uni-programming environment. In the second project you will design and implement appropriate support for multiprogramming. You will extend the system calls to handle process management and inter-process communication primitives. In addition you will implement a priority based scheduling and aging mechanism in nachos. You will add this to the coded first project. Make sure you correct all the deficiencies in your first project before starting the second project.

Phase 1 –Memory allocation for multiprogramming: (15%)

Required Reading –

- **Translate.h / .cc –** Each object of this class is a translation of a single virtual page to a physical page.
- **Addrspace.h / .cc –** Consists of all data needed to keep track of executing user programs. The constructor allocates memory/pages to processes assuming every virtual address is the same as its physical address. This restricts us to running one user program at a time. In this project we modify the constructor to allow multiple user programs to be run concurrently.
- **Progtest.cc –** The StartProcess function serves as the starting point of every thread, where the addrspace is created and execution of the userprogram begins. You might have to create a similar function to account for new threads that you create.
- **System.h / .cc –** All global/system objects are defined here.
- **Machine.h / .cc** - emulates the part of the machine that executes user programs: main memory, processor registers, etc.

To implement multiprogramming we will have to alter Nachos so that each process is maintained in its own system thread. We will have to take care of memory allocation and de-allocation. Here are the details:

1. Add a case in exception handler so that non-system call exceptions can finish (currentThread>Finish()) the thread. This will be important, as a

run time exception should not cause the operating system to shut down.
2. Implement multiprogramming. The code we have given you is restricted to running one user program at a time. You will need to make some changes to addrspace.h, and addrspace.cc in order to convert the system from uniprogramming to multiprogramming. You will need to:

1. Come up with a way of allocating physical memory frames so that multiple programs can be loaded into memory at once.
2. Provide a way of copying data to/from the kernel from/to the user's virtual address space.
3. Properly handling freeing address space when a user program finishes.
4. It is very important to alter the user program loader algorithm such that it handles memory in terms of pages. Currently, memory space allocation assumes that a process is loaded into a contiguous section of memory. Once multiprogramming is active, memory will no longer appear contiguous in nature. If you do not correct the routine, it is most likely that loading another user program will corrupt the operating system.

Phase 2 –Process Management: (35%)

1. Implement the **SpaceID Exec(char *name, int priority)** system call. Exec starts a new user program running within a new system thread with the given priority. You will need to examine the "StartProcess" function in progtest.cc in order to figure out how to set up user space inside a system thread. Exec should return –1 on failure, else it should return the "Process SpaceID" of the user level program it just created. (Note: SpaceIDs can be kept track of in a similar manner to OpenFileIDs of your project 1, except that you will want to keep track of them outside the thread.). For this phase you can simply ignore the priority.
2. Implement the **int Join(SpaceID id) and void Exit(int exitCode)** system calls. Join will wait and block on a "Process SpaceID" as noted in its parameter. Exit returns an exit code to whomever is doing a join. The exit code is 0 if a program successfully completes, another value if

there is an error. The exit code parameter is set via the **exitcode** parameter. Join returns the exit code for the process it is blocking on, -1 if the join fails. A user program can only join to processes that are directly created by **the Exec system call**. You can not join to other processes or to yourself. You will have to use semaphores inside your system calls to coordinate Joining and Exiting user processes. Also make sure that all processes release the resources they have been allocated, after they Exit.

Phase 3 – IPC primitives (Semaphores) (10%)

1. Implement the **int CreateSemaphore(char *name, int semval)** system call. You will have to update start.s and syscall.h to add the new system call signatures. You will create a container at the system level that can hold upto 10 named semaphores. The CreateSemaphore system call will return 0 on success and –1 on failure. The CreateSemaphore system call will fail if there are not enough free spots in the container, the name is null, or the initial semaphore value is less than 0.
2. Implement **int wait(char *name) and int signal(char *name)** system calls. **Make sure you follow the wait and signal as the mnemonics for these two and not down and up or P and V.** The name parameter is the name of the semaphore. Both system calls return 0 on success and –1 on failure. Failure can occur if the user gives an illegal semaphore name (one that has not been created).

Phase 4 –Priority Scheduling & Simple Aging

Required Reading –

- **Thread.h / .cc** – The thread class has methods like Yield, Sleep, Finish to manage the scheduling of threads. Understand of this class is essential to proceed.
- **List.h /.cc** – Implementation of a Generic Linked List. Understand the importance of the methods of this class clearly especially the SortedRemove & SortedInsert. The ready queue maintained by the scheduler is an object of this type.

- **Scheduler.h / .cc** – Implementation of the nachos thread scheduler & dispatcher. You will have to make the majority of your changes in this class.

In this phase you will be implementing a policy that schedules threads depending upon the priority that you set for threads using the Exec(char * name, int priority) syscall. The details are as follows -

- Modify the thread scheduler to always return highest priority thread. You will have to create another parameter in the Thread class– the priority level of the thread represented by an integer value. The range of thread priorities can be found in thread.h. Provide the appropriate tests in order to demonstrate the success of your priority scheduling system. **Note:** Enabling the –rs option for the test programs causes a thread to stop for context switching Yield the CPU to another thread (that could have lower priority) after a given time slice. You might want to have a look at the Thread: :Yield() method to take care of this. **(7%)**

- Most priority scheduling solutions will starve out a low priority thread. After you complete and test the above part, implement a simple aging system to take care of the starvation problem. Under this policy the priority of a thread decreases one unit for every x times the thread is run. That is for every x thread switches from ready to run, decrement the priority of the high priority threads by 1. Specify x as a constant in your system, with the value -1 indicating that aging is disabled. Add thread ("t") debug statements to display the trace of the aging algorithm. Provide the appropriate tests in order to demonstrate the success of your aging system. **(9%)**

Phase 5 –Putting it all together

1. Implement a simple shell program to test your new system calls implemented as above. The shell should take a command at a time, and run the appropriate user program. The shell should "Join" on each program "Exec"ed, waiting for the program to exit. On return from the Join, print the exit code if it is anything other than 0 (normal

execution). Also, design the shell such that it can run program in the background. Any command starting with character (&) should run in the background. (Ex: &create will run the test program create program in the background.) **(6%)**

2. Solve the Dining Philosopher problem discussed in your text book. Use Semaphore you designed for realizing mutual exclusion and synchronization among the philosophers. **(8%)**

Documentation - Includes internal documentation, and external documentation as described:

- How did you maintain a list of all processes in the system? What other data structures did you require? **(3%)**
- Explain the significance of the Join & Exit system calls. How did you synchronize the 2 syscalls? **(4%)**
- What changes did you make to implement Phase 4, and why? **(3%)**

Project 6: Implementing File-System API
Operating Systems One-month project

Objectives:

To enhance Nachos application interface by adding a file system application programming interface (API). This API will include a set of system calls that will allow programmatic calls to Nachos file system from a C application program.

Project Description:

Read and understand components and architecture of Nachos system. See detailed documentation available in Nachos Roadmap. Section 5 of roadmap has details about Nachos implementation of file system. The code for the file systems can be found in the directory nachos-3.4/code/filesys. We have given below a class diagram that shows the various classes and the relationship among them.

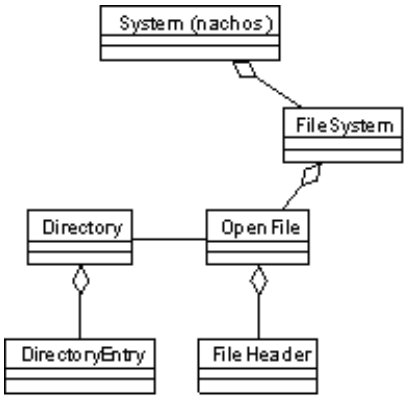


Figure 1: Nachos File System (nachos-3.4/code/filesys/)

Figure 1 shows nachos system (code/threads/system.h, system.cc) instantiating an object of FileSystem class. FileSystem maintains all openfiles (OpenFile objects). A directory object has an Openfile object. It also has a table of objects of DirectoryEntry. All the methods such as read

and write defined in the classes are accessible at kernel level and not at application level. The current Nachos file system is implemented by directly making the corresponding calls to the UNIX file system. The API you build will facilitate operations such as creation, reading, writing, seeking into and deletion of different types of files from an application program. To accomplish this you will make use of the classes nachos provides.

Understand the Code

The first step is to read and understand the existing code. After you expand the tar distribution of nachos, examine the code directory. Then “gmake all” from the code directory to carry out a preliminary compile and link of code in all the directories. Makes sure compilation finishes without errors. (Get help from TAs in case you have errors.) There is a trivial test provided with the distribution code/test directory, ‘halt’; all halt does is to turn around and ask the operating system to shut the machine down. Change directory into userprog directory. Run the command ‘nachos –rs 1023 -x ../test/halt’. This should halt the simulated MIPS machine and type out statistics for that particular run.

Examine the following files to understand Nachos:

code/userprog

syscall.h: This provides the code and function prototype for system calls that user level test programs can invoke.

exception.cc: The handler for system calls and other user-level exceptions, such as page faults. Currently only the ‘halt’ system call is supported.

code/machine

machine.*emulates part of the machine that executes user programs: main memory, processor registers, etc.

mipssim.ccemulates the integer instruction set of a MIPS R2/3000 processor.

console.*emulates a terminal device using UNIX files. A terminal is (i) byte oriented, (ii)incoming bytes can be read and written at the same time, and (iii) bytes arrive asynchronously (as a result of user keystrokes), without being explicitly requested.

code/threads

thread.* implements thread (unit of work) for nachos. Methods to control operation of threads. All the thread specific (local) properties are define here.

system.* implements all the system facilities. The common (global) components of the machine are declared, instantiated here. For example, FileSystem object, currentThread object (pointer to current thread), etc.

synchconsole.* routine to synchronize lines of I/O in Nachos. Use the synchconsole class to ensure that your lines of text from your programs are not intermixed.

code/test/*C programs that will be cross-compiled to MIPS and run in Nachos; start.s has all the assembly language stubs for the system calls.

code/filesys

openfile.ha stub defining the Nachos file system routines.

Read the Makefile in the various directories. In general, while working on Nachos projects you will add classes (your code) to userprog and C programs to test directory. You may modify existing classes in other directories.

The following are the steps in adding and testing a new system call to Nachos:

1. In userprog/syscall.h file, define a code for the system call, and add the C function prototype corresponding to the system call. (Remember it is

C language interface).
2. In start.s add a “macro” stub corresponding to the syscall. This is a set of assembly language instructions and directives that will help compiler substitute the C call with this stub code. See start.s for examples.
3. ExceptonHandler function in exception.cc provides the entry point into kernel for handling the system call. Add the code for exception handler for the new syscall to exception.cc.
4. If you added any new supporting classes in userprog, include them Makefile.common definitions for USERPROG_H, USERPROG_C, and USERPROG_O.
5. Add a C test program (say, trial1.c) that uses this system and change Makefile in test directory to compile and link it. See examples in the current Makefile in the test directory. “gmake”
6. “gmake” in userprog and test the new system call by executing: nachos –rs 5678 –x ../test/trial1

Design of File Syscall API

In order to fully realize how an operating system works, it is important to understand the distinction between kernel (system space) and user space. If we remember from class, each process in a system has its own local information, including program counters, registers, stack pointers, and file system handles. Although the user program has access to many of the local pieces of information, the operating system controls the access. The operating system is responsible for ensuring that any user program request to the kernel does not cause the operating system to crash. The transfer of control from the user level program to the system call occurs through the use of a “system call” or “software interrupt/trap”. Before invoking the transfer from the user to the kernel, any information that needs to be transferred from the user program to the system call must be loaded into the registers of the CPU. For pass by value items, this process merely involves placing the value into the register. For pass by reference items, the value placed into the register is known as a “user space pointer”. Since the user space pointer has no meaning to the kernel, we will have to translate the contents of the user space into the kernel such that we can manipulate the

information. When returning information from a system call to the user space, information must be placed in the CPU registers to indicate either the success of the system call or the appropriate return value.

Our simulator can run normal programs compiled from C -- see the Makefile in the ‘test’ subdirectory for an example. The compiled programs must be linked with some special flags, then converted into Nachos format, using the program “coff2noff” (in bin directory). Floating-point operations are not supported.

Implement exception handling and handle the basic system calls for file system. Figure 2 depicts the role of file syscall API.

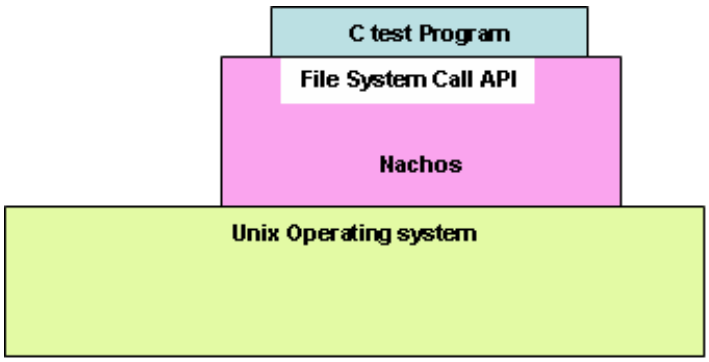
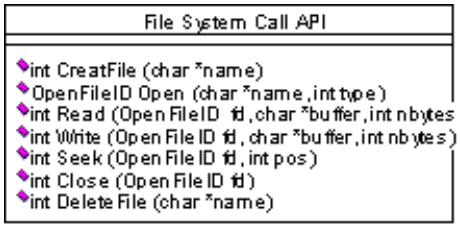


Figure 2: File Sys Call Interface between a user program and Nachos kernel

Signatures of the file system call API are given below. Implement the signatures exactly as given here. OpenFileID is integer type.



You will need to do the following steps:

- Implement the int CreateFile(char *name) and int DeleteFile(char * name) system calls. The createfile system call will use the Nachos Filesystem Object Instance to create a zero length file. Remember, the filename exists in user space. This means the buffer that the user space pointer points to must be translated from user memory space to system memory space. The both system call return 0 for successful completion, -1 for an error.
- Implement the OpenFileID Open(char *name, int type) and int Close(OpenFileID id) system calls. The user program can open three types of “files”: type 1 is read only (RO), type 2 is read and write (RW), and type 3 is read and execute (RX). If the type parameter is set to any other value, the system call should fail. Each process will allocate a fixed size file descriptor table. For now, set this size to be 8 file descriptors. The first two file descriptors, 0 and 1, will be reserved for console input and console output respectively. The open file system call will be responsible for translating the user space buffers when necessary and allocating the appropriate kernel constructs. For the case of actual files, you will use the filesystem objects provided to you in the filesystem directory. (NOTE: We are using the FILESYSTEM_STUB code). Calls will use the Nachos Filesystem Object Instance to open and close files. The Open system call returns the file descriptor id (OpenFileID == an integer number), or –1 if the call fails. Open can fail for several reasons, such as trying to open a file that does not exist or if there is not enough room in the file descriptor table. The close system call will take a file descriptor as the parameter. The system call will return –1 on failure and 0 on success.
- Implement the int Read(OpenFileID fd, char *buffer, int nbytes) and int Write(OpenFileID fd, char *buffer, int nbytes) system calls. These system calls respectively read and write to a file defined by file descriptor fd. Remember, you must translate the character buffers appropriately and you must differentiate between console IO (OpenFileID 0 and 1) and different types of files. The read and write interaction will work as follows:

For console read and write, you will use the SynchConsole class (code/threads directory), instantiated through the gSynchConsole global variable (see threads/system.h,.cc). You will use the default SynchConsole

behaviors for read and write, however you will be responsible for returning the correct types of values to the user. Read and write to Console will return the number of characters read or written. In the case of read or write failure to console, the return value should be –1. If an end of file is reached for a read operation from the console, the return value should be –2. End of file from the console is returned when the user types in Control-A. Read and write for console will use ASCII data for input and output. (Remember, ASCII data is NULL (\0) terminated). Also for file types RO and RW use ASCII reads, and for RX use BINARY read and write.

- Implement the int Seek(OpenFileID fd, int pos) system call. Seek will move the file cursor to a specified location. The parameter pos will be the absolute character position within a file. If pos is a –1, the cursor will be moved to the end of file. The system call will return the actual file position upon success, -1 if the call fails. Seeks on console IO will fail. Seek differs from other calls above since you will have to implement the equivalent Nachos code yourself in the appropriate file in filesys directory.
- Implement a new system call “Delete” int DeleteFile (char *name). Delete will have code 13. You will have to update the Start.s file in the test directory and the syscall.h file in the userprog directory. Recompile and test it with a sample program delete.
- Implement a createFile C user program to test the createfile system call. You are not going to pass command line arguments to the call, so you will have to either use a fixed filename, or prompt the user for one when you have console IO working.
- Implement a help user program. All help does is it prints a list to standard output of all the user programs you are going to create or have created in the test directory. Help should list each program and a brief 1 line description of each program.
- Implement an echo user program. For each line of input from the console, the line gets echoed back to the console.
- Implement a cat user program. Ask for a filename, and display the contents of the file to the console.
- Implement a copy user program. Ask for a source and destination filename and copy the file.

- Implement an outFile user program. This program asks for a destination file, takes input from the console, and writes it to the destination.
- Implement a testSeek user program. This program should demonstrate whether your seek solution works correctly.
- Implement any other tests you feel are necessary to ensure the correctness of your solution. BIG HINT: Implement tests to cover some of the things we changed that are not tested by parts e through k.

NOTE: A large portion of your grade will depend not only on the correctness of your implementation but how accurately your code conforms to the system call specifications presented in this document. As we are building a robust operating system, the user should not be able to perform any system call that will crash Nachos. Make sure there is a default case in the exception handler that will print out a message and halt the system in the case.

Documentation

This includes internal documentation (comments) and a BRIEF, BUT COMPLETE external document (read as: paper) describing what you did to the code and why you made your choices. DO NOT PARAPHRASE THIS LAB DESCRIPTION AND DO NOT TURN IN A PRINTOUT OF YOUR CODE AS THE EXTERNAL DOCUMENTATION.

Deliverables and grading

When you complete your project, remove all executables and object files. If you want me to read a message with your code, create a README.NOW file and place it in the nachos code directory. Tar and compress the code, and submit the file using the online submission system. It is important that you follow the design guidelines presented for the system calls. We will be running my own shells and test programs against your code to determine how accurately you designed your lab, and how robust your answers are. Grading for the implementation portion will depend on how robust and how accurate your solution is. Remember, the user should not be able to do

anything to corrupt the system, and system calls should trap as many error conditions as possible.

Review question: Process
Review Question 1 of Introduction to Operating Systems

Consider a hypothetical 32-bit microprocessor that has 32-bit instructions composed of two fields. The first byte contains the opcode and the remainder an immediate operand or an operand address.

- What is the maximum directly addressable memory capacity (in bytes)?

2(32-8) = 16,777,216 bytes = 16 MB

- Discuss the impact on the system speed if the microprocessor data bus has

- a 32-bit local address bus and a 16-bit local data bus.

Instruction and data transfers would take three bus cycles each - one for the address and two for the data.

- a 16-bit local address bus and a 16-bit local data bus.

Instruction and data transfers would take four bus cycles each - two for the address and two for the data.

- How many bits are needed for the program counter and the instruction register?

24 bits for the PC (24-bit addresses), 32 bits for the IR (32-bit addresses)

In virtually all systems that include DMA modules, DMA access to main memory is given higher priority than processor access to main memory. Why?

If a processor is held up in attempting to read or write memory, usually no damage occurs except a slight loss of time. However, a DMA transfer may

be to or from a device that is receiving or sending data in a stream (e.g., disk or network), and cannot be stopped. Thus, if the DMA module is held up (denied continuing access to main memory), data will be lost.

Why does a batch system need memory protection? Why does a multiprogrammed batch system need memory management and scheduling? Why do both types of systems need interrupts?

A batch system needs memory protection so that programs can't modify the monitor.

A multiprogrammed batch system needs memory management so that multiple jobs can be kept in memory at the same time. Memory management will determine which portion of memory each job can use. The system also needs a scheduling algorithm to determine which job can run at any given time.

Both types of systems need interrupts so that the operating system can regain control of the CPU. A batch system needs this to time-out a long job and a multiprogrammed system needs this to share the processor among the active processes.

What is the purpose of system calls, and how do system calls relate to the operating system and to the concept of dual-mode (kernel mode and user mode) operation?

With a time sharing system, the primary concern is turnaround time. A round-robin scheduler would given every process a chance to run on the CPU for a short time, and reduce the average turnaround time. If the scheduler instead let one job run until completion, then the first job would have a short turnaround time, but later ones would have to wait for a long time.

In a batch system, the primary concern is throughput. In this case, the time spent switching between jobs is wasted, so a more efficient scheduling

algorithm would be first-come, first-served, and let each job run on the processor as long as it wants.

In IBM's mainframe operating system, OS/390, one of the major modules in the kernel is the System Resource Manager (SRM). This module is responsible for the allocation of resources among address spaces (processes). The SRM gives OS/390 a degree of sophistication unique among operating systems. No other mainframe operating system, and certainly no other type of operating system, can match the functions performed by SRM. The concept of resource includes processor, real memory and I/O channels. SRM accumulates statistics pertaining to utilization of processor, channel and various key data structures. Its purpose is to provide optimum performance based on performance monitoring and analysis. The installation sets forth various performance objectives, and these serve as guidance to the SRM, which dynamically modifies installation and job performance characteristics based on system utilization. In turn, the SRM provides reports that enable the trained operator to refine the configuration and parameter settings to improve user service.

This problem concerns one example of SRM activity. Real memory is divided into equal-sized blocks called frames, of which there may be many thousands. Each frame can hold a block of virtual memory referred to as a page. SRM receives control approximately 20 times per second and inspects each and every page frame. If the page has not been referenced or changed, a counter is incremented by 1. Over time, SRM averages these numbers to determine the average number of seconds that a page frame in the system goes untouched. What might be the purpose of this and what action might SRM take?

The system operator can review this quantity to determine the degree of "stress" on the system. By reducing the number of active jobs allowed on

the system, this average can be kept high. A typical guideline is that this average should be kept above 2 minutes [IBM86]. This may seem like a lot, but it isn't.

What main advantage does a system with virtual memory have over a system without it? How does virtual memory help with memory protection?

With virtual memory, a process image does not need to be entirely in physical memory for it to run. In addition, it is not restricted to the size of physical memory. The process can address as much memory as it wants, and the operating system brings blocks from secondary storage into physical memory as they are needed. Virtual memory also helps with memory protection because one process cannot address memory that belongs to another process. All addresses are mapped by the operating system into its own address space.

What is the difference between a symmetric multiprocessing system and a distributed operating system?

In a symmetric multiprocessing system, a group of processors with the same capabilities share memory and are located within the same computer system. The existence of multiple processors is transparent to the user. In a distributed operating system, the group of processors each addresses its own memory, are located in different computers connected by a network, and can have differing capabilities. The operating system allows the user to treat the collection of systems as one computer.

What shortcoming does the two-state process model have? How does the Blocked state fix this problem?

In the two-state process model, if a process is interrupted for I/O, it is placed in a queue with other processes that are ready to run. When the operating system wants to find a new process to run on the CPU, it must search through the queue for a process that is ready. Using a model with a

Blocked state means that there are two separate queues: one for ready processes and one for those that are waiting for I/O. When the operating system wants to find a process that is ready, it can simply take the first process on the ready queue.

What shortcoming does the five-state process model have? How does swapping solve this problem?

Assuming virtual memory is not being used, then only a limited number of processes can be active at any time. In the five-state process model, it is possible that all active processes are waiting on an I/O event and none of them are in the ready queue. In this case, the CPU will be idle.

Swapping can solve this problem by taking a process out of memory and writing it out to disk, then bringing in a new process that can run on the CPU. The five-state model is augmented with a new state called Suspended to indicate the process is swapped out.

What is contained in a process image? Some older operating systems require the entire process image to be stored in main memory while the process is executing. What problem does this create and how is it solved?

The process image contains user data (for program data and the stack), the user program, the system stack (for procedure calls), and the process control block (process identifiers, processor state, various types of control information).

If the entire image must be in main memory, then the image must be small and only a limited number of processes can be active at any time. This problem can be solved by using virtual memory, so that only a portion of the process image (the portion currently being used) needs to be in memory at any given time.

It states that UNIX is unsuitable for real-time applications because a process executing in kernel mode may not be preempted. Elaborate.

Because there are circumstances under which a process may not be preempted (i.e., it is executing in kernel mode), it is impossible for the operating system to respond rapidly to real-time requirements.

Use the ps program to list all the processes running on a Linux machine (including system processes and processes for all users). List one process for each unique state that is listed (i.e. one per unique entry in the state column) and describe what state this process is in. See the manual page for help with ps. Next, create five new processes in quick succession and leave them running. Using ps, list their process identifiers. Is there any relation between the identifiers they are assigned?

You can use ps ax to list all the processes on a Linux machine. Here is a list of some of the processes I found:

PID TTY STAT TIME COMMAND

1 ? S 0:04 init [3]

2 ? SN 0:00 [ksoftirqd/0]

3 ? S< 0:02 [events/0]

4 ? S< 0:00 [khelper]

9 ? S< 0:00 [kthread]

18 ? S< 0:06 [kacpid]

118 ? S< 0:00 [kblockd/0]

176 ? S 0:00 [pdflush]

179 ? S< 0:00 [aio/0]

178 ? S 0:01 [kswapd0]

771 ? S 0:04 [kseriod]

829 ? S< 0:00 [ata/0]

843 ? S< 0:00 [reiserfs/0]

983 ? Ss 0:00 /sbin/devfsd /dev

5192 ? S 0:00 [khubd]

6328 ? Ss 0:00 metalog [MASTER]

6332 ? S 0:00 metalog [KERNEL]

6366 ? Ss 0:00 /usr/sbin/acpid -c /etc/acpi/events

6991 ? Ss 0:00 /usr/sbin/cupsd

7326 ? Ss 0:00 /usr/sbin/fcron

7422 ? Ss 0:00 /usr/sbin/speedfreqd -P /var/run/speedfreq.pid -p pow

There are many more. The TTY column indicates whether the process is attached to a termanl, and STAT gives the state of the process. Most processes are sleeping (S), one has been given a low priority (N), and others have been given a high priority (<). Notice that typically important system processes have a high priority.

If you create five processes in quick succession, they will typically have sequential process identifiers.

How is a process switch different from a mode switch?

A process switch involves the operating system having to do a lot of work, whereas a mode switch is done completely in hardware. A mode switch involves saving the state of the current process (in hardware), changing to supervisor mode, then giving control to the operating system. The idea is that the operating system will execute a short handler, then return control to the processor. The processor will restore the state of the original process and continue.

With a process switch, the operating system must give control of the CPU to a different process, rather than resuming with the previous process that was running. Thus it must save the context of the current process in its PCB, update the PCB state and accounting information, move the PCB to a new queue, execute the scheduling algorithm to choose a new process, remove the PCB of that process from the ready queue, update its memory management structures, and restore its context into the processor.

In a threaded program, which resources are shared by the entire process and which are associated with each thread?

The process address space, structures used for interprocess communication, files, and I/O resources are associated with the entire process. Each thread gets its own execution state, context (registers), stack, and local variables.

Consider the following code:

1. for (i = 0; i < 10; i++)
2. for (j = 0; j < 10; j++)
3. a[i] = a[i] * j

    - Give one example of the spatial locality in the code.

Spatial locality occurs when the array a is accessed sequentially.

- Give one example of the temporal locality in the code.

Temporal locality occurs when i is used repeatedly with the same value in the second loop.

Consider a memory system with the following parameters:

- Tc = 100 ns
- Tm = 1200 ns
- Cc = 0.01 cents/bit
- Cm = 0.001 cents/bit

1. What is the cost of 1 MByte of main memory?

Cost = Cm x 8 x 220 = $83.89

1. What is the cost of 1 Mbyte of main memory using cache memory technology?

Cost = Cc x 8 x 220 = $838.86

1. If the effective access time is 10% greater than the cache access time, what is the hit ratio H?

1.1 x Tc = Tc + (1 - H)Tm110 = 100 + (1 - H) x 12001200 x H = 1190H = 1190/1200 = 99.2%

A computer has a cache, main memory, and a disk used for virtual memory. If a referenced word is in the cache, 20 ns are required to access it. If it is in main memory but not in the cache, 60 ns are needed to load it into the cache (this includes the time to originally check the cache), and then the reference is started again. If the word is not in main memory, 12 ms are required to fetch the word from disk, followed by 60 ns to copy it to the cache, and then the reference is started again. The cache hit ratio is 0.9 and the main memory hit ratio is 0.6. What is the average time in ns required to access a referenced word on this system?

There are three cases to consider:

Location of Word	Probability	Total Time for Access (ns)
In cache	0.9	20
Not in cache, but in main memory	(0.1)(0.6) = 0.06	60 + 20 = 80
Not in cache or main memory	(0.1)(0.4) = 0.04	12ms + 60 + 20 = 12,000,080

So the average access time would be:

Avg = (0.9)(20) + (0.06)(80) + (0.04)(12000080) = 480,026 ns

Suppose that we have a multi-programmed computer in which each job has identical characteristics. In one computation period, T, for a job, half the time is spent in I/O and the other half in processor activity. Each job runs for a total of N periods. Assume that simple round-robin scheduling is used, and that I/O operations can overlap with processor operation. Define the following quantities:

- Turnaround time = actual time to complete a job.
- Throughput = average number of jobs completed per time period T
- Processor utilization = percentage of time that the processor is active (waiting)

Compute these quantities for one, two, and four simultaneous jobs, assuming that the period T is distributed in each of the following ways:

1. I/O first half, processor second half

When there is one job, it can do I/O or run on the processor whenever it wants. So the quantities are:

- Turnaround Time = N*T
- Throughput = 1/N
- Processor Utilization = 50%

When there are two jobs, one starts right away and does I/O. When it switches to run on the CPU, the second can start its I/O. This delays the second job for 1/2*N, but otherwise they alternate between I/O and CPU. Assume the jobs are long, so the extra 1/2 a cycle is insignificant. Then:

- Turnaround Time = N*T
- Throughput = 2/N
- Processor Utilization = 100%

When there are 4 jobs, the CPU is round-robin among the four, as is the I/O. This means the jobs are interleaved as:

Job1: I/O CPU I/O CPU

Job2: I/O CPU I/O CPU

Job3: I/O CPU I/O CPU

Job4: I/O CPU I/O CPU

A job can execute for one cycle T, then it must wait for T before doing another cycle. Again assume the jobs are long so that any initial wait is insignificant. Then:

- Turnaround Time = (2N-1)*T
- Throughput = 2/N
- Processor Utilization = 100%
- I/O first and fourth quarters, processor second and third quarter

The answers for this part are the same as the first. This is easy to see for the case of 1 job and 2 jobs. When there are 4 jobs, the CPU is round-robin among the four, as is the I/O. This means the jobs are interleaved as:

Job1: I CP O I CP O

Job2: I CP O I CP O

Job3: I CP O I CP O

Job4: I CP O I CP O

An I/O-bound program is one that, if run alone, would spend more time waiting for I/O than using the processor. A processor-bound program is the opposite. Suppose a short-term scheduling algorithm favors those programs that have used little processor time in the recent past.

- Explain why this algorithm favors I/O-bound programs.

The algorithm favors I/O bound processes because these will be much less likely to have used the processor recently. As soon as they are done with an I/O burst, they will get to execute on the processor.

- Explain why this algorithm does not permanently deny processor time to processor-bound programs.

This algorithm does not permanently deny processor time to processor-bound programs, because once they have been prevented from using the processor for some time they will be favored by the scheduling algorithm.

Review question: Synchronization
Review question of Synchronization

Explain the three different types of scheduling: long-term, mid-term, and short-term. What does the operating system do and when?

Long-term scheduling is used to admit new processes to the system and is done whenever a user wants to create a new process.

Medium-term scheduling is used to swap processes between secondary storage and main memory and is done periodically to ensure the CPU utilization is high and that the page fault rate is low.

Short-term scheduling is used to allocate the CPU to a process and is done whenever a process has its timeslice expire, or when the current process needs to perform some I/O.

Explain why Shortest Process Next, Shortest Remaining Time, and Highest Response Ratio Next all need to estimate the future CPU bursts of each process. Explain how this is done.

All of these scheduling algorithms try to order the queue according to the next CPU burst a process is likely to have. Since it is not possible to look into the future or examine the process and know the length of the next burst, this quantity must be estimated. This is done by using past CPU bursts to predict future burst. The estimate of the next CPU burst is calculated as a weighted, exponential average. The previous CPU burst is given weight alpha and the current estimate is given weight (1 - alpha). By varying alpha, more weight can be given to the most recent CPU burst, allowing for faster reaction to changes in a process but also a highly varying estimate. Giving weight to the previous estimate allows the estimate to change more smoothly. In effect, the estimation equation gives an exponentially smaller weight to CPU bursts the farther in the past they occurred.

Most round-robin schedulers use a fixed size quantum. Give an argument in favor of a small quantum. Now give an argument in favor of a large quantum. Compare and contrast the types of systems and jobs to which the arguments apply. Are there any for which both are reasonable?

A small quantum reduces the response time for all processes, which is important for interactive processes. However, a long quantum reduces the overhead of process switching, which increases throughput and CPU utilization. A short quantum is useful for a general-purpose computer. A long quantum is useful for a batch system. You might have a system that uses a short quantum when there are jobs that need a quick response time, then lengthens the quantum when heavy computation needs to be done.

Which type of process is generally favored by a multilevel feedback queuing scheduler -- a processor-bound process or an I/O-bound process? Briefly explain why.

A I/O bound process is favored by a multilevel feedback queueing scheduler because jobs that use the CPU heavily will be moved to low priority queues. This will leave the I/O bound processes in the higher priority queues.

Problems

Consider an environment in which there is a one-to-one mapping between user-level threads and kernel-level threads that allows one or more threads within a process to issue blocking system calls while other threads continue to run. Explain why this model can make multithreaded programs run faster than their single-threaded counterparts on a uniprocessor machine.

The issue here is that a program spends much of its time waiting for I/O to complete. In a multithreaded program, one KLT can make the blocking system call, while the other KLTs can continue to run. On a uniprocessor machine, a process that would otherwise have to block for all these calls can continue to run its other threads.

A multiprocessor with eight processors has 20 attached tape drives. There is a large number of jobs submitted to the system, and each requires a maximum of four tape drives to complete execution. Assume that each job starts running with only three tape drives for a long period before requiring the fourth tape drive for a short period toward the end of its operation. Also assume an endless supply of such jobs.

a. Assume the scheduler in the OS will not start a job unless there are four tape drives available. When a job is started, four drives are assigned immediately and are not released until the job finishes. What is the maximum number of jobs that can be in progress at once? What is the maximum and minimum number of tape drives that may be left idle as a result of this policy?

If a conservative policy is used, at most 20/4 = 5 processes can be active simultaneously. Because one of the drives allocated to each process can be idle most of the time, at most 5 drives will be idle at a time. In the best case, none of the drives will be idle.

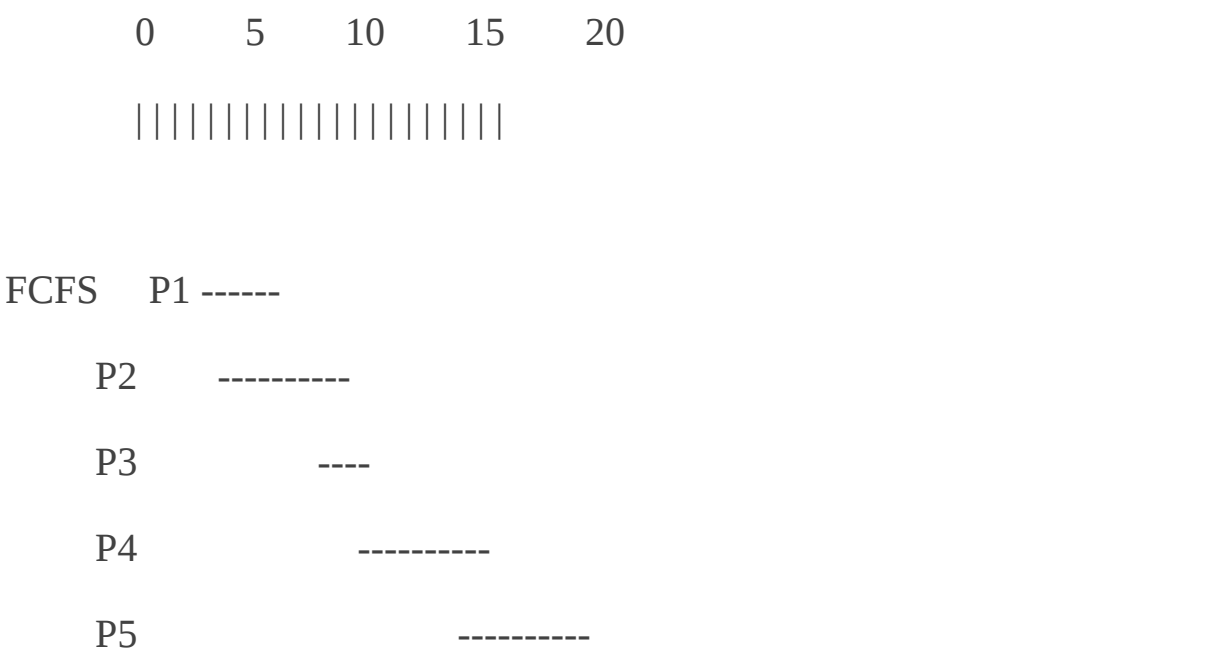
b. Suggest an alternative policy to improve tape drive utilization and at the same time avoid system deadlock. What is the maximum number of jobs that can be in progress at once? What are the bounds on the number of idling tape drives?

To improve drive utilization, each process can be initially allocated with three tape drives. The fourth one will be allocated on demand. In this policy, at most floor(20/3) = 6 processes can be active simultaneously. The minimum number of idle drives is 0 and the maximum number is 2.

Consider the following set of processes:

Process Name	Arrival Time	Processing Time
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5

Perform the same analysis as depicted for this set.



RR q=1 P1 -- -- --

P2 -- -- -- -- --

P3 -- --

P4 -- -- -- -- --

P5 -- -- -- ----

RR q=4 P1 ------

P2 -------- --

P3 ----

P4 -------- --

P5 -------- --

SPN P1 ------

P2 ----------

P3 ----

P4 ----------

P5 ----------

SRT P1 ------

P2 ----------

P3 ----

P4 ----------

P5 ----------

HRRN P1 ------

P2 ----------

P3 ----

P4 ----------

P5 ----------

FB q=1 P1 -- -- --

P2 -- -- ---- --

P3 -- --

P4 -- -- -- -- --

P5 -- -- -- ----

FB q=2^i P1 -- ----

P2 -- ---- ----

P3 -- --

P4 ------ ----

P5 -- --------

For feedback queueing with a quantum of 1, I assume that when a process is taken off the processor and placed in a lower priority queue, it is not eligible to go right back onto the processor. It must wait another quantum. For feedback queueing with a quantum of 2i, I assume that a newly arriving process can't preempt the current process until it is done with its quantum. This is different from the book's assumption and therefore different from Figure 9.5 in the book.

Here are the metrics:

	1	2	3	4	5	
Ta	0	1	3	9	12	
Ts	3	5	2	5	5	

==========

FCFS	Tf	3	8	10	15	20
	Tq	3	7	7	6	8	6.20
	Tq/Ts	1.00	1.40	3.50	1.20	1.60	1.74

==========

RR q=1	Tf	6	11	8	18	20
	Tq	6	10	5	9	8	7.60
	Tq/Ts	2.00	2.00	2.50	1.80	1.60	1.98

==========

RR q=4	Tf	3	10	9	19	20
	Tq	3	9	6	10	8	7.20

Tq/Ts 1.00 1.80 3.00 2.00 1.60 1.88

==========

SPN Tf 3 10 5 15 20

Tq 3 9 2 6 8 5.60

Tq/Ts 1.00 1.80 1.00 1.20 1.60 1.32

==========

SRT Tf 3 10 5 15 20

Tq 3 9 2 6 8 5.60

Tq/Ts 1.00 1.80 1.00 1.20 1.60 1.32

==========

HRRN Tf 3 10 5 15 20

Tq 3 7 7 6 8 6.20

Tq/Ts 1.00 1.40 3.50 1.20 1.60 1.74

==========

FB q=1 Tf 7 11 6 18 20

Tq 7 10 3 9 8 7.40

Tq/Ts 2.33 2.00 1.50 1.80 1.60 1.85

==========

FB q=1 Tf 4 10 8 16 20

Tq 4 9 5 7 8 7.00

Tq/Ts 1.33 1.80 2.50 1.40 1.60 1.81

==========

Assume the following burst-time pattern for a process: 6,4,6,4,13,13,13, and assume that the initial guess is 10.

Five batch jobs, A through E, arrive at a computer center at essentially the same time. They have an estimated running time of 15, 9, 3, 6, and 12 minutes, respectively. Their (externally defined) priorities are 6, 3, 7, 9, and 4 respectively, with a lower value corresponding to a higher priority. For each of the following scheduling algorithms, determine the turnaround time for each process and the average turnaround time for all jobs. Ignore process switching overhead. Explain how you arrived at your answers. In the last three cases, assume that only one job at a time runs until it finishes and that all jobs are completely processor bound.

Process	Running Time	Priority
A	15	4
B	9	7
C	3	3
D	6	1
E	12	6

a. round robin with a time quantum of 1 minute

1 2 3 4 5 Elapsed Time

==========

A B C D E 5

A B C D E 10

A B C D E 15

A B D E 19

A B D E 23

A B D E 27

A B E 30

A B E 33

A B E 36

A E 38

A E 40

A E 42

A 43

A 44

A 45

Process Turnaround Time

==========

A 45

B 35

C 13

D 26

E 42

Average Turnaround Time = 32.2 minutes

b. priority scheduling

Process Priority Turnaround Time

==========

B 7 9

E 6 21

A 4 36

C 3 39

D 1 45

Average Turnaround Time = 30 minutes

c. FCFS (run in order 15, 9, 3, 6, and 12)

Process Turnaround Time

==========

A 15

B 24

C 27

D 33

E 45

Average Turnaround Time = 28.8 minutes

d. shortest job first

Process Running Time Turnaround Time

==========

C 3 3

D 6 9

B 9 18

E 12 30

A 15 45

Average Turnaround Time = 21 minutes

Consider a set of three periodic tasks with the execution profiles as follows:

Process	Arrival Time	Execution Time	Ending Deadline
A(1)	0	10	20
A(2)	20	10	40
.	.	.	.
.	.	.	.
.	.	.	.
B(1)	0	10	50
B(2)	50	10	100
.	.	.	.
.	.	.	.
.	.	.	.
C(1)	0	15	50
C(2)	50	15	100
.	.	.	.

.	.	.	.
.	.	.	.

Develop scheduling diagrams for this set of tasks.

Suppose that a microcontroller used in an automobile has four priority levels for interrupts. The units interrupting the controller include an impact sensor subsystem, which needs attention with 0.1 ms, along with four other subsystems as follows:

Subsystem	Interrupt Rate	Max service time (ms)
Fuel/ignition	500/sec	1
Engine temperature	1/sec	100
Dashboard display	800/sec	0.2
Air conditioner	1/sec	100

Discuss how priorities could be assigned to guarantee the response time of 0.1 ms for the impact sensor and to handle each of the other critical units before the next interrupt is produced.

The impact sensor must be given the highest priority because its required response time is less than the service time for each of the other interrupts types; no other subsystem can share this highest priority.

The air conditioner can be given the lowest priority level because its function is not safety-critical.

We have two priority levels left for three subsystems.

Because the engine temperature monitor has a maximum service time of 100 ms it must have a lower priority than the display and fuel/ignition subsystems, which need several hundreds of service periods per second.

Finally, the display and fuel/ignition subsystems can share the same priority level since each can be served after the other without exceeding the time available before the next interrupt.

For example, if the fuel/ignition subsystem service starts right before the display interrupt, a total of 1.2 ms will elapse before the service routines of both interrupts are completed, whereas there is 1/800 sec - 1.25 ms available before the next dashboard display interrupt arrives.

Note that we have ignored the service time for impact sensor subsystem interrupts that have the highest priority because they occur very infrequently, and when they do, all else becomes unimportant.

Review question: Memory management
Review question of Memory management

Why does fixed partitioning suffer from internal fragmentation whereas dynamic partitioning suffers from external fragmentation? When is compaction needed?

Fixed partitioning suffers from internal fragmentation because some processes may use less memory than the fixed partition size. Dynamic partitioning suffers from external fragmentation because the memory "holes" left between two partitions may be too small for another process to use. Compaction is used in a system with dynamic partitioning to place all of the partitions in a contiguous range of memory. The effect of this is to combine all the small holes into one larger hole.

What improvements does paging make relative to fixed partitioning?

With paging, a process is divided into a large number of small, fixed size pages. These pages are then placed into frames of main memory, each of which is the size of a page. The main improvement over fixed partitioning is that the pages of the process do not need to be contiguous in main memory. In addition, internal fragmentation is reduced because there is only fragmentation on the last page of the process and the pages are small. The cost of these improvements is the overhead of needing a page table for each process.

How is a logical address translated into a physical address on a system that uses pure paging? How is the translation done for a system that uses pure segmentation?

In a system that uses pure paging, the logical address is divided into a page number and an offset. The page number is used as an index into the page table for the running process; the corresponding entry contains the frame

number that the page is mapped to. The offset is added to the frame number to form a physical address.

In a system that uses pure segmentation, the logical address is divided into a segment number and an offset. The segment number is used as an index into the segment table for the running process; the corresponding entry contains a segment base and length. The segment base is the beginning of the segment of memory that the process is mapped into and the length gives the length of this segment. If the offset is greater than the segment length, a segmentation error occurs. Otherwise, the offset is added to the segment base to form a physical address.

Why is locality of reference so important for a virtual memory system?

Locality is important because a virtual memory system keeps only a small subset of a process image in memory at any given time. If locality holds, then many memory references will refer to this subset and the process will be able to run normally. However, if locality doesn't hold, then memory references will not map to a valid physical address. When this happens, a memory fault occurs and the missing data will need to be loaded from secondary storage. The process will run slowly because secondary storage is much slower than main memory.

When does a page fault occur? Describe what the operating system does to handle a page fault.

A page fault occurs when a reference is mapped to a page that is not currently in main memory. When this happens, the process is suspended and the operating system takes control. If main memory is full, then a page must be replaced. Once an empty frame is found, the operating system schedules

an I/O operation to bring in the page. After the page is brought in, and the page table updated, the process can be started again.

What is thrashing and how can an operating system take steps to avoid it?

Thrashing occurs when a system spends more time swapping than running a process. The operating system can avoid thrashing by reducing the number of processes that are in memory, i.e. by swapping some out.

What are the advantages of having a fixed or a variable resident set? For both of these methods of sizing the resident set, explain what it means to have a global or a local replacement policy.

The advantage of having a fixed resident set means that no other process can take frames away from you. However, this has to take into account the maximum need of a process and so it may waste memory. The advantage of having a dynamic resident set means the number of frames dedicated to a process can grow or shrink over time, more closely matching its needs. Of course, there is more overhead involved with this approach.

A fixed size resident set implies a local replacement policy. Any time a process needs a new page, one of its own pages must be replaced.

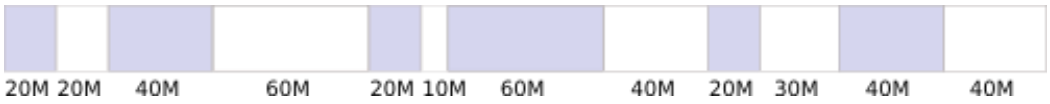
Having a variable resident set allows the operating system to do either local or global replacement. Doing local replacement means that the process always replaces one of its own frames and the resident set size can be increased or decreased separately by the operating system. Doing global replacement means the process can automatically increase its resident set by choosing to replace someone else's page with its own. It may also have its resident set decreased when someone else chooses one of its pages.

What is the advantage of precleaning versus demand cleaning?

Precleaning may decrease the page fault rate if the operating system is able to bring in pages in advance of them being used by a process. This may be particularly effective at the time the process is first loaded.

Problems

A dynamic partitioning scheme is being used, and the following is the memory configuration at a given point in time:



The shaded areas are allocated blocks; the white areas are free blocks. The next three memory requests are for 40M, 20M, and 10M. Indicate the starting address for each of the three blocks using the indicated placement algorithms:

1. First-fit

* The 40 M block fits into the second hole, with a starting address of 80M.

* The 20M block fits into the first hole, with a starting address of 20M.

* The 10M block is placed at location 120M.

2. Best-fit

The three starting addresses are 230M, 20M, and 160M, for the 40M, 20M, and 10M blocks, respectively.

3. Next-fit. Assume the most recently added block is at the beginning of memory.

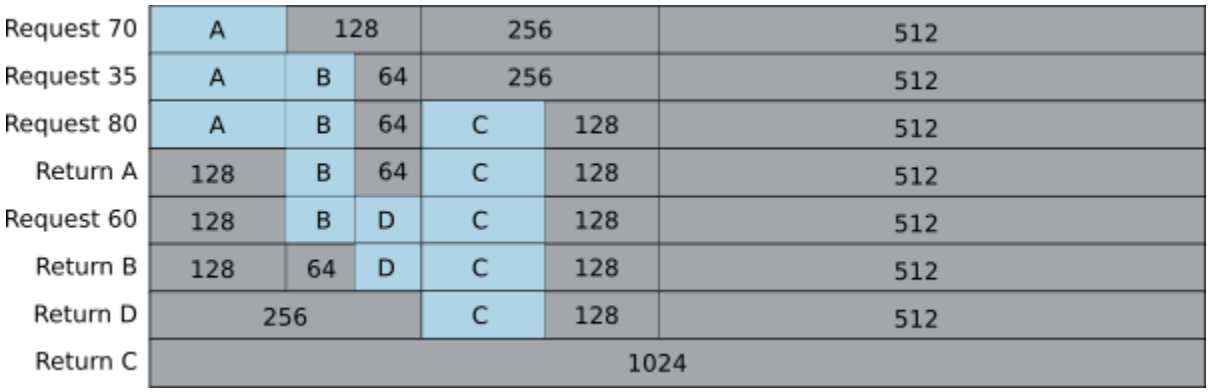
The three starting addresses are 80M, 120M, and 160M, for the 40M, 20M, and 10M blocks, respectively.

4. Worst-fit

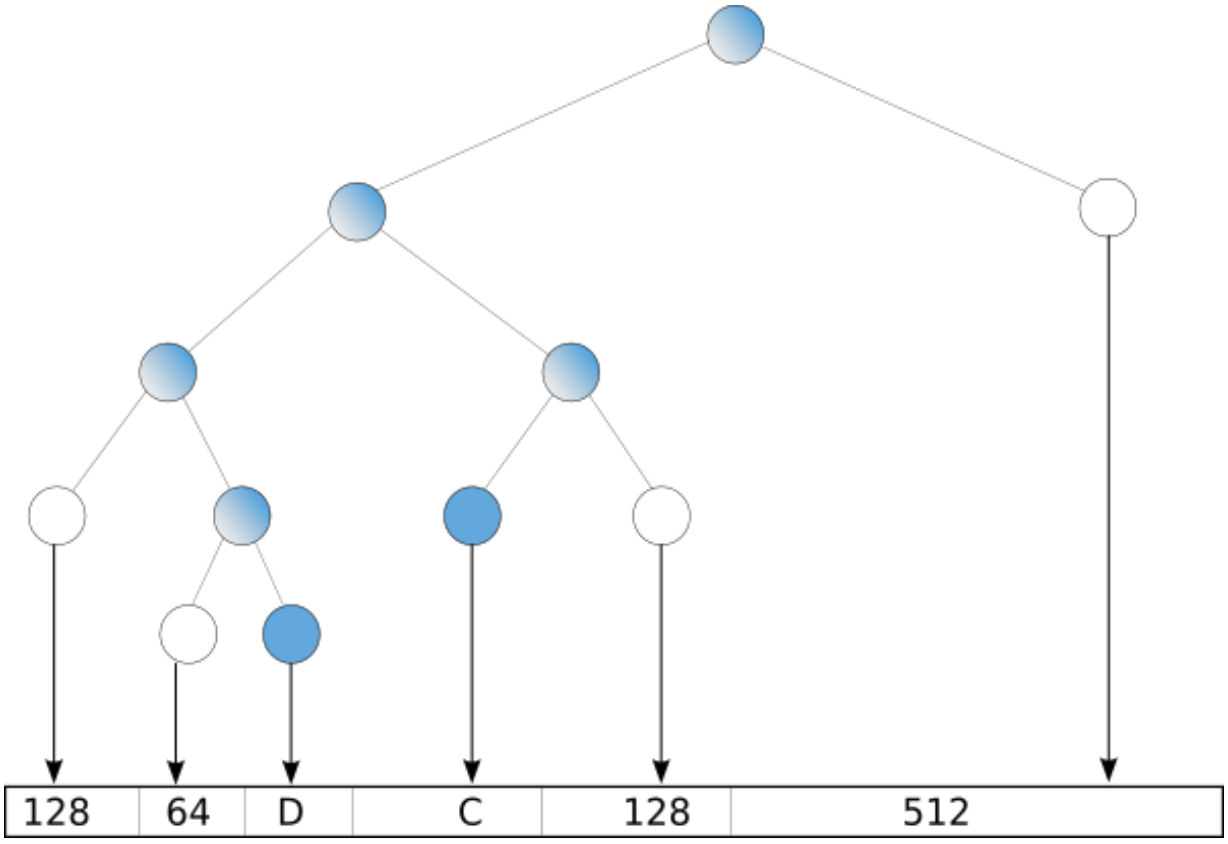
The three starting addresses are 80M, 230M, and 360M, for the 40M, 20M, and 10M blocks, respectively.

A 1-Mbyte block of memory is allocated using the buddy system.

a. Show the results of the following sequence in a figure similar to Figure 7.6: Request 70; Request 35; Request 80; Return A; Request 60; Return B; Return D; Return C.



b. 2. Show the binary tree representation following Return B.



Consider a buddy system in which a particular block under the current allocation has an address of 011011110000.

a. If the block is of size 4, what is the binary address of its buddy?

If the block is of size 4, the binary address of its buddy is 011011110100

b. 2. If the block is of size 16, what is the binary address of its buddy?

If the block is of size 16, the binary address of its buddy is 011011100000.

Consider a simple paging system with the following parameters: 232 bytes of physical memory; page size of 210 bytes; 216 pages of logical address space.

a. How many bits are in a logical address?

The number of bytes in the logical address space is (216 pages) * (210 bytes/page) = 226 bytes. Therefore, 26 bits are required for the logical address.

b. How many bytes in a frame?

A frame is the same size as a page, 210 bytes.

c. How many bits in the physical address specify the frame?

The number of frames in main memory is (232 bytes of main memory)/(210 bytes/frame) = 222 frames. So 22 bits is needed to specify the frame.

d. How many entries in the page table?

There is one entry for each page in the logical address space. Therefore there are 216 entries.

e. How many bits in each page table entry? Assume each page table entry includes a valid/invalid bit.

In addition to the valid/invalid bit, 22 bits are needed to specify the frame location in main memory, for a total of 23 bits.

Consider a paged virtual memory system with 32-bit virtual addresses and 1KB pages. Each page table entry requires 32 bits. It is desired to limit the page table size to one page.

a. How many levels of page tables are required?

Virtual memory can hold (232 bytes of main memory)/(210 bytes/page) = 222 pages, so 22 bits are needed to specify a page in virtual memory. Each page table contains (210 bytes per page table)/(4 bytes/entry) = 28 entries. Thus, each page table can handle 8 of the required 22 bits. Therefore, 3 levels of page tables are needed.

b. What is the size of the page table at each level? Hint: One page table size is smaller.

Tables at two of the levels have 28 entries; tables at one level have 26 entries. (8 + 8 + 6 = 22).

c. The smaller page size could be used at the top level or the bottom level of the page table hierarchy. Which strategy consumes the least number of pages?

Less space is consumed if the top level has 26 entries. In that case, the second level has 26 pages with 28 entries each, and the bottom level has 214 pages with 28 entries each, for a total of 1 + 26 + 214 pages = 16,449 pages. If the middle level has 26 entries, then the number of pages is 1 + 28 + 214 pages = 16,641 pages. If the bottom level has 26 entries, then the number of tables is 1 + 28 + 216 pages = 65,973 pages.

A process has four page frames allocated to it. (All the following numbers are decimal, and everything is numbered starting from zero.) The time of the last loading of a page into each page frame, the last access to the page in each page frame, the virtual page number in each page frame, and the referenced (R) and modified (M) bits for each page frame are as shown (the times are in clock ticks from the process start at time 0 to the event -- not the number o fticks since the event to the present).

Virtual Page Number	Page Frame	Time Loaded	Time Referenced	R Bit	M Bit
2	0	60	161	0	1
1	1	130	160	1	0
0	2	26	162	1	0
3	3	20	163	1	1

A page fault to virtual page 4 has occured at time 164. Which page frame will have its contents replaced for each of the following memory management policies? Explain why in each case.

a. FIFO (first-in-first-out)

Frame 3 since it was loaded the longest ago at time 20.

b. LRU (least recently used)

Frame 1 since it was referenced the longest ago at time 160.

c. Clock

Clear R in Frame 3 (oldest loaded), clear R in Frame 2 (next oldest loaded), victim Frame is 0 since R=0.

d. Optimal (Use the following reference string.)

Replace the page in Frame 3 since the virtual page number 3 (in Frame 3) is used furthest in the future.

e. Given the aforementioned state of memory just before the page fault, consider the following virtual page reference string: 4,0,0,0,2,4,2,1,0,3,2. How many page faults would occur if the working set policy with LRU were used with a window size of 4 instead of a fixed allocation? Show clearly where each fault would occur.

There are 6 faults, indicated by *:

(Window) - {working set}

(1 2 0 3) - {2 1 0 3}

(2 0 3 4) - {2 0 3 4}*

(0 3 4 0) - {0 3 4}

(3 4 0 0) - {0 3 4}

(4 0 0 0) - {0 4}

(0 0 0 2) - {0 2}*

(0 0 2 4) - {0 2 4}*

(0 2 4 2) - {0 2 4}

(2 4 2 1) - {2 4 1}*

(4 2 1 0) - {2 4 1 0}*

(2 1 0 3) - {2 1 0 3}*

(1 0 3 2) - {2 1 0 3}

Consider a system with memory mapping done on a page basis and usin ga single-level page table. Assume that hte necessary page table is always in memory.

a. If a memory reference takes 200 ns, how long does a paged memory reference take?

400 nanoseconds. 200 to get the page table entry, and 200 to access the memory location

.

b. Now we add an MMU that imposes an overhead of 20 ns on a hit or a miss. If we assume that 85% of all memory references hit in the MMU TLB, what is the Effective Memory Access Time (EMAT)?

There are two cases. First, when the TLB contains the entry required, we have 20 ns overhead on top of the 200 ns memory access time. Second, when the TLB does not contain the item, we have an additional 200 ns to get the required entry into the TLB:

(220 * 0.85) + (420 * 0.15) = 250 ns

c. Explain how the TLB hit rate affects the EMAT.

The higher the TLB hit rate is, the smaller the EMAT is, because the additional 200 ns penalty to get the entry into the TLB contributes less to the EMAT.

Assume that a task is divided into four equal-sized segments and that the system builds an eight-entry page descriptor table for each segment. Thus, the system has a combination of segmentation and paging. Assume also that the page size is 2 Kbytes.

a. What is the maximum size of each segment?

(8 entries in the page table) x 2K = 16K.

b. What is the maximum logical address space for the task?

(16 K) x 4 segments per task = 64K.

c. Assume that an element in physical location 00021ABC is accessed by this task. What is the format of the logical address that the task generates for it? What is the maximum physical address space for the system?

We know the offset is 11 bits since the page size is 2K. The page table for each segment has eight entries, so it needs 3 bits. That leaves 2 bits for the segment number. So the format of the address is 2 bits for segment number, 3 bits for page number, and 11 bits for offset.

The physical address is 32 bits wide total, so the frame number must be 21 bits wide. Thus 00021ABC is represented in binary as:

Frame Offset

0000 0000 0000 0010 0001 1 | 010 1011 1100

The maximum physical address space is 232 = 4 GB.

Note that the virtual address space is only 16 bits wide: 11 for the offset, 3 for the page number, and 2 for the segment number. This means a process cannot logically address the entire 4GB space ... it is limited to only 64K.

Assume that we have a demand-paged memory with the page table held in registers. It takes 8 milliseconds to service a page fault if an empty frame is available or if the replaced page is not modified, and 20 milliseconds if the replaced page is

modified. Assume the page to be replaced is modified 70 percent of the time. Memory access time is 100 nanoseconds.

What is the maximum acceptable page-fault rate if we want the system to have an effective memory access time of 200 nanoseconds?

EMAT = 200ns = X(100ns) + (1-X)(.70(20ms + 100ns) + .3(8ms + 100ns))

= 100X + (1-X)(16.4ms)

= 100X - 16,400,000X + 16,400,000

X = 16399800/163999000

= .9999939024

Therefore the maximum acceptable page-fault rate is (1-X) = .00061%.

Review question: IO devices and File systems
Review question of IO devices and File systems

Why does fixed partitioning suffer from internal fragmentation whereas dynamic partitioning suffers from external fragmentation? When is compaction needed?

Fixed partitioning suffers from internal fragmentation because some processes may use less memory than the fixed partition size. Dynamic partitioning suffers from external fragmentation because the memory "holes" left between two partitions may be too small for another process to use. Compaction is used in a system with dynamic partitioning to place all of the partitions in a contiguous range of memory. The effect of this is to combine all the small holes into one larger hole.

What improvements does paging make relative to fixed partitioning?

With paging, a process is divided into a large number of small, fixed size pages. These pages are then placed into frames of main memory, each of which is the size of a page. The main improvement over fixed partitioning is that the pages of the process do not need to be contiguous in main memory. In addition, internal fragmentation is reduced because there is only fragmentation on the last page of the process and the pages are small. The cost of these improvements is the overhead of needing a page table for each process.

How is a logical address translated into a physical address on a system that uses pure paging? How is the translation done for a system that uses pure segmentation?

In a system that uses pure paging, the logical address is divided into a page number and an offset. The page number is used as an index into the page table for the running process; the corresponding entry contains the frame

number that the page is mapped to. The offset is added to the frame number to form a physical address.

In a system that uses pure segmentation, the logical address is divided into a segment number and an offset. The segment number is used as an index into the segment table for the running process; the corresponding entry contains a segment base and length. The segment base is the beginning of the segment of memory that the process is mapped into and the length gives the length of this segment. If the offset is greater than the segment length, a segmentation error occurs. Otherwise, the offset is added to the segment base to form a physical address.

Why is locality of reference so important for a virtual memory system?

Locality is important because a virtual memory system keeps only a small subset of a process image in memory at any given time. If locality holds, then many memory references will refer to this subset and the process will be able to run normally. However, if locality doesn't hold, then memory references will not map to a valid physical address. When this happens, a memory fault occurs and the missing data will need to be loaded from secondary storage. The process will run slowly because secondary storage is much slower than main memory.

When does a page fault occur? Describe what the operating system does to handle a page fault.

A page fault occurs when a reference is mapped to a page that is not currently in main memory. When this happens, the process is suspended and the operating system takes control. If main memory is full, then a page must be replaced. Once an empty frame is found, the operating system schedules

an I/O operation to bring in the page. After the page is brought in, and the page table updated, the process can be started again.

What is thrashing and how can an operating system take steps to avoid it?

Thrashing occurs when a system spends more time swapping than running a process. The operating system can avoid thrashing by reducing the number of processes that are in memory, i.e. by swapping some out.

What are the advantages of having a fixed or a variable resident set? For both of these methods of sizing the resident set, explain what it means to have a global or a local replacement policy.

The advantage of having a fixed resident set means that no other process can take frames away from you. However, this has to take into account the maximum need of a process and so it may waste memory. The advantage of having a dynamic resident set means the number of frames dedicated to a process can grow or shrink over time, more closely matching its needs. Of course, there is more overhead involved with this approach.

A fixed size resident set implies a local replacement policy. Any time a process needs a new page, one of its own pages must be replaced.

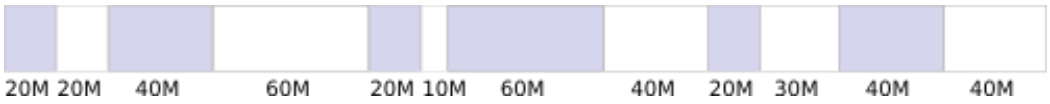
Having a variable resident set allows the operating system to do either local or global replacement. Doing local replacement means that the process always replaces one of its own frames and the resident set size can be increased or decreased separately by the operating system. Doing global replacement means the process can automatically increase its resident set by choosing to replace someone else's page with its own. It may also have its resident set decreased when someone else chooses one of its pages.

What is the advantage of precleaning versus demand cleaning?

Precleaning may decrease the page fault rate if the operating system is able to bring in pages in advance of them being used by a process. This may be particularly effective at the time the process is first loaded.

Problems

A dynamic partitioning scheme is being used, and the following is the memory configuration at a given point in time:



The shaded areas are allocated blocks; the white areas are free blocks. The next three memory requests are for 40M, 20M, and 10M. Indicate the starting address for each of the three blocks using the indicated placement algorithms:

1. First-fit

* The 40 M block fits into the second hole, with a starting address of 80M.

* The 20M block fits into the first hole, with a starting address of 20M.

* The 10M block is placed at location 120M.

2. Best-fit

The three starting addresses are 230M, 20M, and 160M, for the 40M, 20M, and 10M blocks, respectively.

3. Next-fit. Assume the most recently added block is at the beginning of memory.

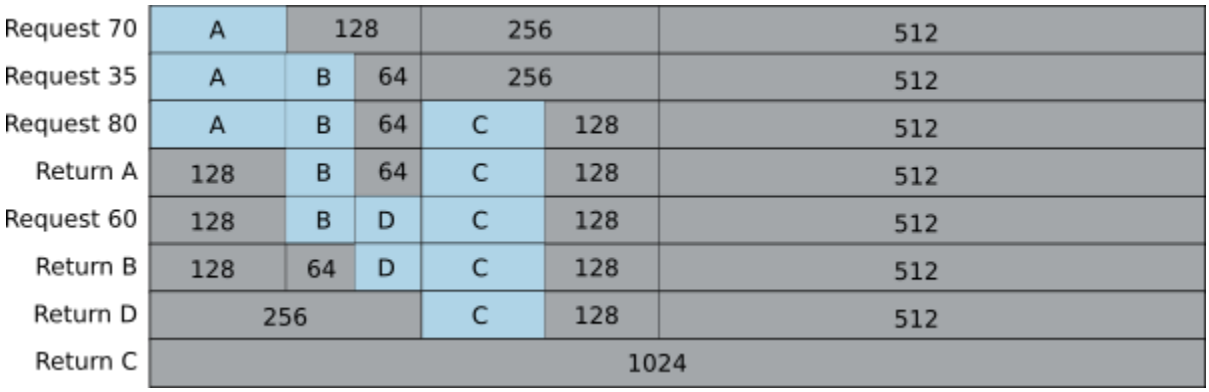
The three starting addresses are 80M, 120M, and 160M, for the 40M, 20M, and 10M blocks, respectively.

4. Worst-fit

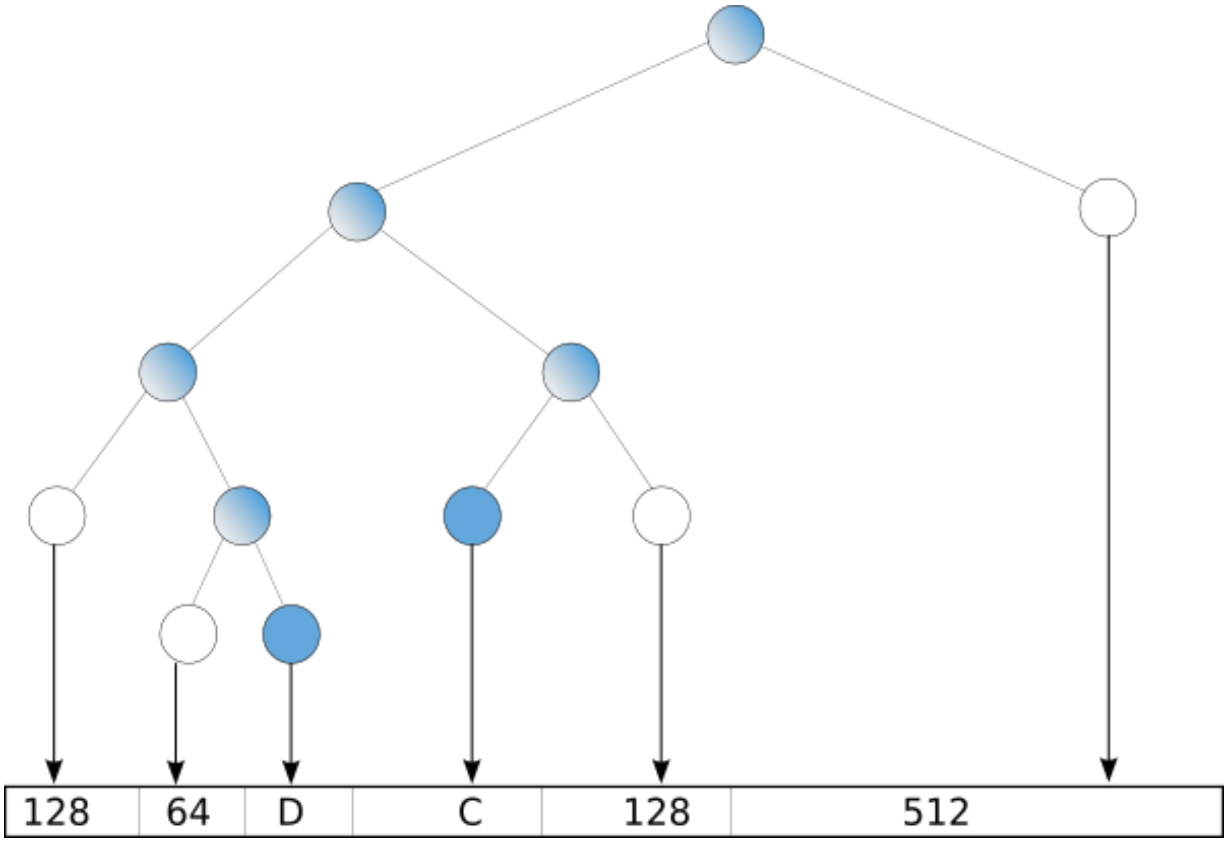
The three starting addresses are 80M, 230M, and 360M, for the 40M, 20M, and 10M blocks, respectively.

A 1-Mbyte block of memory is allocated using the buddy system.

a. Show the results of the following sequence in a figure similar to Figure 7.6: Request 70; Request 35; Request 80; Return A; Request 60; Return B; Return D; Return C.



b. 2. Show the binary tree representation following Return B.



Consider a buddy system in which a particular block under the current allocation has an address of 011011110000.

a. If the block is of size 4, what is the binary address of its buddy?

If the block is of size 4, the binary address of its buddy is 011011110100

b. 2. If the block is of size 16, what is the binary address of its buddy?

If the block is of size 16, the binary address of its buddy is 011011100000.

Consider a simple paging system with the following parameters: 232 bytes of physical memory; page size of 210 bytes; 216 pages of logical address space.

a. How many bits are in a logical address?

The number of bytes in the logical address space is (216 pages) * (210 bytes/page) = 226 bytes. Therefore, 26 bits are required for the logical address.

b. How many bytes in a frame?

A frame is the same size as a page, 210 bytes.

c. How many bits in the physical address specify the frame?

The number of frames in main memory is (232 bytes of main memory)/(210 bytes/frame) = 222 frames. So 22 bits is needed to specify the frame.

d. How many entries in the page table?

There is one entry for each page in the logical address space. Therefore there are 216 entries.

e. How many bits in each page table entry? Assume each page table entry includes a valid/invalid bit.

In addition to the valid/invalid bit, 22 bits are needed to specify the frame location in main memory, for a total of 23 bits.

Consider a paged virtual memory system with 32-bit virtual addresses and 1KB pages. Each page table entry requires 32 bits. It is desired to limit the page table size to one page.

a. How many levels of page tables are required?

Virtual memory can hold (232 bytes of main memory)/(210 bytes/page) = 222 pages, so 22 bits are needed to specify a page in virtual memory. Each page table contains (210 bytes per page table)/(4 bytes/entry) = 28 entries. Thus, each page table can handle 8 of the required 22 bits. Therefore, 3 levels of page tables are needed.

b. What is the size of the page table at each level? Hint: One page table size is smaller.

Tables at two of the levels have 28 entries; tables at one level have 26 entries. (8 + 8 + 6 = 22).

c. The smaller page size could be used at the top level or the bottom level of the page table hierarchy. Which strategy consumes the least number of pages?

Less space is consumed if the top level has 26 entries. In that case, the second level has 26 pages with 28 entries each, and the bottom level has 214 pages with 28 entries each, for a total of 1 + 26 + 214 pages = 16,449 pages. If the middle level has 26 entries, then the number of pages is 1 + 28 + 214 pages = 16,641 pages. If the bottom level has 26 entries, then the number of tables is 1 + 28 + 216 pages = 65,973 pages.

A process has four page frames allocated to it. (All the following numbers are decimal, and everything is numbered starting from zero.) The time of the last loading of a page into each page frame, the last access to the page in each page frame, the virtual page number in each page frame, and the referenced (R) and modified (M) bits for each page frame are as shown (the times are in clock ticks from the process start at time 0 to the event -- not the number o fticks since the event to the present).

Virtual Page Number	Page Frame	Time Loaded	Time Referenced	R Bit	M Bit
2	0	60	161	0	1
1	1	130	160	1	0
0	2	26	162	1	0
3	3	20	163	1	1

A page fault to virtual page 4 has occured at time 164. Which page frame will have its contents replaced for each of the following memory management policies? Explain why in each case.

a. FIFO (first-in-first-out)

Frame 3 since it was loaded the longest ago at time 20.

b. LRU (least recently used)

Frame 1 since it was referenced the longest ago at time 160.

c. Clock

Clear R in Frame 3 (oldest loaded), clear R in Frame 2 (next oldest loaded), victim Frame is 0 since R=0.

d. Optimal (Use the following reference string.)

Replace the page in Frame 3 since the virtual page number 3 (in Frame 3) is used furthest in the future.

e. Given the aforementioned state of memory just before the page fault, consider the following virtual page reference string: 4,0,0,0,2,4,2,1,0,3,2. How many page faults would occur if the working set policy with LRU were used with a window size of 4 instead of a fixed allocation? Show clearly where each fault would occur.

There are 6 faults, indicated by *:

(Window) - {working set}

(1 2 0 3) - {2 1 0 3}

(2 0 3 4) - {2 0 3 4}*

(0 3 4 0) - {0 3 4}

(3 4 0 0) - {0 3 4}

(4 0 0 0) - {0 4}

(0 0 0 2) - {0 2}*

(0 0 2 4) - {0 2 4}*

(0 2 4 2) - {0 2 4}

(2 4 2 1) - {2 4 1}*

(4 2 1 0) - {2 4 1 0}*

(2 1 0 3) - {2 1 0 3}*

(1 0 3 2) - {2 1 0 3}

Consider a system with memory mapping done on a page basis and usin ga single-level page table. Assume that hte necessary page table is always in memory.

a. If a memory reference takes 200 ns, how long does a paged memory reference take?

400 nanoseconds. 200 to get the page table entry, and 200 to access the memory location

.

b. Now we add an MMU that imposes an overhead of 20 ns on a hit or a miss. If we assume that 85% of all memory references hit in the MMU TLB, what is the Effective Memory Access Time (EMAT)?

There are two cases. First, when the TLB contains the entry required, we have 20 ns overhead on top of the 200 ns memory access time. Second, when the TLB does not contain the item, we have an additional 200 ns to get the required entry into the TLB:

(220 * 0.85) + (420 * 0.15) = 250 ns

c. Explain how the TLB hit rate affects the EMAT.

The higher the TLB hit rate is, the smaller the EMAT is, because the additional 200 ns penalty to get the entry into the TLB contributes less to the EMAT.

Assume that a task is divided into four equal-sized segments and that the system builds an eight-entry page descriptor table for each segment. Thus, the system has a combination of segmentation and paging. Assume also that the page size is 2 Kbytes.

a. What is the maximum size of each segment?

(8 entries in the page table) x 2K = 16K.

b. What is the maximum logical address space for the task?

(16 K) x 4 segments per task = 64K.

c. Assume that an element in physical location 00021ABC is accessed by this task. What is the format of the logical address that the task generates for it? What is the maximum physical address space for the system?

We know the offset is 11 bits since the page size is 2K. The page table for each segment has eight entries, so it needs 3 bits. That leaves 2 bits for the segment number. So the format of the address is 2 bits for segment number, 3 bits for page number, and 11 bits for offset.

The physical address is 32 bits wide total, so the frame number must be 21 bits wide. Thus 00021ABC is represented in binary as:

Frame Offset

0000 0000 0000 0010 0001 1 | 010 1011 1100

The maximum physical address space is 232 = 4 GB.

Note that the virtual address space is only 16 bits wide: 11 for the offset, 3 for the page number, and 2 for the segment number. This means a process cannot logically address the entire 4GB space ... it is limited to only 64K.

Assume that we have a demand-paged memory with the page table held in registers. It takes 8 milliseconds to service a page fault if an empty frame is available or if the replaced page is not modified, and 20 milliseconds if the replaced page is

modified. Assume the page to be replaced is modified 70 percent of the time. Memory access time is 100 nanoseconds.

What is the maximum acceptable page-fault rate if we want the system to have an effective memory access time of 200 nanoseconds?

EMAT = 200ns = X(100ns) + (1-X)(.70(20ms + 100ns) + .3(8ms + 100ns))

= 100X + (1-X)(16.4ms)

= 100X - 16,400,000X + 16,400,000

X = 16399800/163999000

= .9999939024

Therefore the maximum acceptable page-fault rate is (1-X) = .00061%.

Review question: Protection and Security
Review question of Protection and Security

Briefly explain how single, double, and circular buffering can be used to improve the performance of a process.

With single buffering, the operating system dedicates a buffer to a process. This allows the process to work on one block while the I/O system reads another block into the buffer.

With double buffering, the operating system dedicates two buffers to a process. The process can then empty one buffer while the I/O system fills the other. This allows the process to process data as fast as the I/O system can read it in.

Why is the average search time to find a record in a file less for an indexed sequential file than for a sequential file?

With a sequential file, the records are stored in order of a particular key, but searching requires going through all of the records sequentially until a match is found. In an indexed sequential file, the index contains pointers to every K records in the file. For a file with N records, searching for a record with a particular key requires searching only the N/K entries in the index, and then the K records in that section of the file. Depending on the size of K, this can be considerably less than sequentially searching through all N entries.

Briefly describe the four categories of information stored in a file directory. Give an example of an item from each category, along with a one sentence description.

- Basic information is used to identify the file and includes the file name and type.
- Address information is used to find the file on disk and includes the address and size of the file.
- Access control information is used to provide sharing and protection and includes the owner and related access information.
- Usage information is used for accounting and includes such items as the date of creation and last modification.

Problems

Perform the same type of analysis as that of Table 11.2 for the following sequence of disk track requests: 27, 129, 110, 186, 147, 41, 10, 64, 120. Assume that the disk head is initially positioned over track 100 and is moving in the direction of decreasing track number. Do the same analysis, but now assume that the disk head is moving in the direction of increasing track number.

Answer:

FIFO		SSTF		SCAN		C-SCAN	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed

27	73	110	10	64	36	64	36
129	102	120	10	41	23	41	23
110	19	129	9	27	14	27	14
186	76	147	18	10	17	10	17
147	39	186	39	110	100	186	176
41	106	64	122	120	10	147	39
10	31	41	23	129	9	129	18
64	54	27	14	147	18	120	9
120	56	10	17	186	39	110	10
Average	61.8	Average	29.1	Average	29.6	Average	38

If the disk head is initially moving in the direction of increasing track number, the results change for only SCAN and C-SCAN:

SCAN		C-SCAN	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed
110	10	110	10
120	10	120	10
129	9	129	9
147	18	147	18
186	39	186	39
64	122	10	176
41	23	27	17
27	14	41	14
10	17	64	23
Average	29.1	Average	35.1

Consider the disk system described in Problem 11.9 and assume the disk rotates at 360 rpm. A processor reads one sector from the disk using interrupt-driven I/O, with one interrupt per byte. If it takes 2.5 μs to process each interrupt, what percentage of time will the processor spend handling I/O (disregard seek time)? Repeat using DMA, and assume one interrupt per sector.

Answer:

If you have the disk rotating at 360 rpm, then one byte is read off the disk in:

T = b/rN = 1/(360 rpm*512 bytes/sector * 96 sectors/track)

T = 3.4 μs

If one interrupt occurs every byte, then the operating system will process an interrupt (2.5 μs), then do some work, then process another interrupt, etc. The time between interrupts is equal to the time it takes to read one byte. In other words, the disk will read one byte, then interrupt the operating system, then read another byte. The operating system will only be able to get work done if it has enough time to finish processing an interrupt before the next one occurs. In this case, the processor will spend 2.5 μs / 3.4 μs or 74% of its time handling I/O.

With DMA, there is only one interrupt for the entire sector. It takes 3.4 μs/byte * 512 bytes/sector = 1,741 μs to read the sector. Therefore the processor only spends 2.5 μs / 1741 μs = .14% of the time handling I/O.

Note: The above answer does not take into account rotational delay. If you include rotational delay, then the time to read a byte becomes very large:

T = b/rN + 1/2r = 83 ms

This means the time the processor spends processing interrupts is very small (approximately .003% in both cases).

It should be clear that disk striping can improve the data transfer rate when the strip size is small compared to the I/O request size. It should also be clear that RAID 0 provides improved performance relative to a single large disk, because multiple I/O requests can be handled in parallel. However, in this latter case, is disk striping necessary? That is, does disk striping improve I/O request rate performance compared to a comparable disk array without striping?

Answer:

Consider a disk array that does not use striping. If multiple I/O requests are issued for files on different disks, then performance is the same as a striped disk array. However, the advantage of striping is that it improves the chances that multiple I/O requests will in fact be for data on different disks. Without striping, all the files from a given directory will be on the same disk, so a single user operating in a single directory will be using one disk instead of all the disks in the array. Likewise, it is likely that all user files will be on one disk, and all binaries on another, so that multiple users accessing their files will use the same disk. Disk striping is a simple, automated way for the administrator of the array to ensure that files are spread out across all the disks.

You purchase a disk drive that spins at 9600 RPM, stores 300 KB per track, and has an average seek time of 8 ms.

- What is the average transfer time to read a 1 MB file, assuming the file is stored on consecutive tracks?

T = Ts + 1/(2r) + b/(rN)

T = .008 + (60 s/m)(1 r)/ (2*9600 r/m) + (1 MB)(1024 KB/MB)(60 s/m)/(9600 r/m)(300 KB/r)

T = .008 s + .003125 s + .021333 s

T = 32.458 ms

- What is the average transfer time to read a 1 MB file, assuming that the file is stored in 10 separate locations on the disk, and each location has 100 KB of contiguous data?

T = 10*(Ts + 1/(2r) + 1/(rN))

T = 10(.008 s + .003125 s + (100 KB)(60 s/m)/(9600 rpm)(300 KB/r)

T = 10(.008 s + .003125 s + .002083 s)

T = 132.083 ms

What file organization would you choose to maximize efficiency in terms of speed of access, use of storage space, and ease of updating (adding/deleting/modifying) when the data are:

- updated infrequently and accessed frequently in random order?

Both the indexed file organization and the hashed file are efficient for frequent access to random parts of a file. Since the file is updated infrequently, the overhead of keeping indexes is reduced.

- updated frequently and accessed in its entirety relatively frequently?

The indexed sequential file is efficient when access is usually to the entire file in sequential order. Keeping multiple indexes adds unnecessary overhead and the hash structure is not as useful for sequential access.

- updated frequently and accessed frequently in random order?

The hashed file is efficient for frequent updates, and also is efficient for random access.

Consider a hierarchical file system in which free disk space is kept in a free space list.

- Suppose the pointer to free space is lost. Can the system reconstruct the free space list?

Yes, it is easy to recover the lost space. Keep a bit map for every block on the disk, initially set to zero. Then traverse the file system starting at the root, and mark the bit mask with a 1 for every block that is used by every file. In the end, those blocks still marked by a zero are free.

- Suggest a scheme to ensure that the pointer is never lost as a result of a single memory failure.

One solution is to keep a copy of the free space pointer in several different places on the disk.

Consider the organization of a UNIX file as represented by the inode (Figure 12.13). Assume that there are 12 direct block pointers in a singly, doubly, and triply indirect pointer in each inode. Further, assume that the system block size and the disk sector size are both 8K. If the disk block pointer is 32 bits, with 8 bits to identify the physical disk and 24 bits to identify the physical block, then

- What is the maximum file size supported by this system?

The maximum file size can be found by calculating the space all the different types of block pointers can reference. First we need to calculate the number of pointers an indirect block can hold:

N = 8*1024*8/32 = 2048

Then:

Size = 12*8 + 2048*8 + 2048*2048*8 + 2048*2048*2048*8 KB = 64 TeraBytes

- What is the maximum file system partition supported by this system?

The maximum file system partition is essentially equal to the size of a disk. Since we use 24 bits to address the blocks on each disk, the maximum size of a disk is 128 GB.

- Assuming no information other than that the file inode is already in main memory, how many disk accesses are required to access the byte in position 13,423,956?

The address given is in the 13 MB range. The 12 direct pointers cover 96K, so the address is not located there. The singly indirect pointer covers the next 16 MB of the file, so the address is in a block referenced by the singly indirect pointer. This means we will need two disk accesses, one for the indirect block and another for the block containing the data.

What services are provided by TCP that are not provided by UDP? Briefly explain what each of these services does.

- reliability: TCP ensures that whenever a source host sends a message, it will be received by the destination host in the order it was sent. If the network drops or corrupts the packet, TCP will retransmit it until it is received correctly.
- flow control: TCP checks how much data the destination host can buffer, then makes sure the source host does not send more than this amount at a time. This keeps the source host from sending data faster than the destination host can receive it.
- congestion control: TCP continually adapts to current network conditions, and slows down the source host if the network becomes congested. This keeps the source from sending data faster than the network can deliver it.

Explain how UDP traffic can "take over" the network when competing with TCP traffic.

UDP does not use congestion control, so all it has to do is send at whatever rate it desires. If this causes packet loss due to congestion in a router (a FIFO queue overflowing), then the TCP flows will slow down. If UDP sends at a high enough rate, it can force the TCP flows to slow down enough that they get nothing.

Why does a UDP server need only one socket but a TCP server needs one socket for each client?

A UDP server needs only one socket because anyone can send to it on that socket. It does not establish connections to each client. A TCP server, on the other hand, needs a socket for each client because it establishes a separate connection for each one.